Studies in Computational Intelligence 590

Indrajit Chakrabarti
Kota Naga Srinivasarao Batta
Sumit Kumar Chatterjee

Motion Estimation for Video Coding

Efficient Algorithms and Architectures

Springer

Studies in Computational Intelligence

Volume 590

Series editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland
e-mail: kacprzyk@ibspan.waw.pl

About this Series

The series "Studies in Computational Intelligence" (SCI) publishes new developments and advances in the various areas of computational intelligence—quickly and with a high quality. The intent is to cover the theory, applications, and design methods of computational intelligence, as embedded in the fields of engineering, computer science, physics and life sciences, as well as the methodologies behind them. The series contains monographs, lecture notes and edited volumes in computational intelligence spanning the areas of neural networks, connectionist systems, genetic algorithms, evolutionary computation, artificial intelligence, cellular automata, self-organizing systems, soft computing, fuzzy systems, and hybrid intelligent systems. Of particular value to both the contributors and the readership are the short publication timeframe and the world-wide distribution, which enable both wide and rapid dissemination of research output.

More information about this series at http://www.springer.com/series/7092

Indrajit Chakrabarti · Kota Naga Srinivasarao Batta
Sumit Kumar Chatterjee

Motion Estimation for Video Coding

Efficient Algorithms and Architectures

Springer

Indrajit Chakrabarti
Department of Electronics and ECE
Indian Institute of Technology Kharagpur
Kharagpur
India

Kota Naga Srinivasarao Batta
Department of Electronics and ECE
Indian Institute of Technology Kharagpur
Kharagpur
India

Sumit Kumar Chatterjee
Department of Electronics
and Communication Engineering
National Institute of Technology Sikkim
Ravangla
India

ISSN 1860-949X ISSN 1860-9503 (electronic)
Studies in Computational Intelligence
ISBN 978-3-319-14375-0 ISBN 978-3-319-14376-7 (eBook)
DOI 10.1007/978-3-319-14376-7

Library of Congress Control Number: 2014959195

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media (www.springer.com)

Dedicated to our Families
and Friends

Preface

Overview

A video signal, which is composed of a sequence of still frames of pixels, contains significant amount of redundant information in spatial and temporal domain. Elimination of this redundant information is achieved by efficient video compression methods. To provide interoperability between video encoder and decoder, MPEG-X (1, 2 and 4) and H.26X (261, 262, 263, 264 and 265) video coding standards have been defined by the ITU-G and VCEG. To reduce the temporal redundancy between adjacent frames, the majority of the existing coding standards have adopted Block Matching Algorithms (BMA) for Motion Estimation (ME). BMA calculates motion vector for an entire block of pixels instead of individual pixels. The same motion vector is applicable to all the pixels in the block. This reduces the computational requirement and also results in a more accurate motion vector since the objects are typically a cluster of pixels.

In general, out of all the components of a video encoder, the ME module consumes the major share of overall power. A very simple arithmetic computation is required for ME. However, frequent memory access associated with ME affects the overall speed of operation and the power consumption. The present work has therefore focused on design and development of VLSI architectures for several fast ME architectures characterized by high processing speed, low power, and low area making them suitable for portable video application devices that are typically operated by battery power and involve real time operation.

Organization and Features

This book primarily focuses on low-power VLSI implementation of ME architectures and efficient data reuse technique along with other techniques that have been used to make a high performance ME architecture. In addition, the concept of

scalable video coding based on in-band motion compensated temporal filtering has also been presented.

Chapter 1 gives a brief introduction to the concept of video compression and motion estimation. Chapter 2 provides the background of ME and different fast search techniques for motion estimation and a brief survey of the literature related to the scalable video coding. Chapter 3 explains the design of VLSI architecture for realizing Fast Three Step Search algorithm (FTSS). Chapter 4 explains the implementation of VLSI architecture for Successive Elimination algorithm (SEA). Chapter 5 provides details of fast ME based on a combination of Diamond Search and 1-bit transformation and its architecture. Chapter 6 introduces a new two stage fast algorithm for Variable Block Size Motion Estimation (VBSME) based on pixel truncation and its low power architecture. Chapter 7 gives the fundamentals of Scalable Video Coding based on In-band Motion Compensated Temporal Filtering (IB-MCTF). Finally, Chap. 8 presents a few suggestions for extensions of the present work.

Programs have been developed in Matlab and Verilog to implement the research ideas discussed in depth from Chap. 3 through Chap. 7. Some of these programs have been provided in the two appendices of this book for the benefit of the reader.

Audience

This book presents material that is appropriate for courses at the senior undergraduate level and graduate level in the areas of Video processing and VLSI architectures. It is also suitable for research students who are working on design of VLSI architectures for Video processing applications. Practicing engineers in the area of hardware implementation of video CODEC will also find the book to be immensely useful. Basic familiarity with logic design and hardware description languages is considered adequate to follow the material presented in this book.

Acknowledgments

The authors owe a word of thanks to many people who have helped in various ways in this project. The authors thank their families and friends for their unstinting support. Thanks are due to Professor A.S. Dhar of Department of Electronics and Electrical Communication Engineering, Indian Institute of Technology Kharagpur, India for providing us encouragement. The authors express their sincere gratitude to Dr. Thomas Ditzinger, Springer editor, and Professor Janusz Kacprzyk, series editor for Studies in Computational Intelligence series, for providing us with the necessary support to see our efforts of writing this book come to fruition. The authors thank the anonymous reviewers for their comments. We acknowledge all the authors of

books and research papers that have been referenced while writing this book. A detailed list of references has been provided at the end of each chapter. The authors also acknowledge the support of Indian Institute of Technology Kharagpur.

Kharagpur, November 2014

Indrajit Chakrabarti
Kota Naga Srinivasarao Batta
Sumit Kumar Chatterjee

Contents

About the Authors

Indrajit Chakrabarti received the Bachelor's and Master's degree (in Electronics and Telecommunication) from Jadavpur University, India in 1987 and 1990 respectively. He got Ph.D. degree from Indian Institute of Technology (IIT) Kharagpur, India in 1997. From 1998 to 2004, he worked as an Assistant Professor and later as an Associate Professor in the Department of Electronics and Communication Engineering, IIT Guwahati. Since December 2004 till date, he has been serving as an Associate Professor in the Department of Electronics and Communication Engineering, IIT Kharagpur. His research interests include VLSI architectures for image and video processing, digital signal processing, error control coding and wireless communication. He has published more than 15 papers in international journals, and is a member of IEEE.

Kota Naga Srinivasarao Batta received the Master's degree (in Visual Information and Embedded Systems Engineering) from Indian Institute of Technology (IIT) Kharagpur, India in 2008. Since December 2012 till date, he is a research scholar in the Department of E & ECE, Indian Institute of Technology (IIT) Kharagpur, India. From 2002 to 2006, he worked as an Assistant Professor and from 2008 to 2012 he worked as an Associate Professor in the Department of Electronics and Communication Engineering, Gudlavalleru Engineering College, Gudlavalleru, Andhra Pradesh, India. His research interests include VLSI architectures for image and video compression, digital signal processing and design of embedded systems.

Sumit Kumar Chatterjee received the Master's degree (in Microwaves) from Burdwan University, India in 1999. He got Ph.D. degree from Indian Institute of Technology (IIT) Kharagpur, India in 2011. From 2011 to 2014, he worked as an Assistant Professor in the Department of Electronics and Communication Engineering, Asansol Engineering college. Since September 2014 till date, he has been serving as an Ad hoc Faculty in the Department of Electronics and Communication Engineering, NIT Sikkim. His research interests include VLSI architectures for image and video compression, digital signal processing and error control coding.

Abstract

Video data consist of a time sequence of image frames, and there exists a significant redundancy in temporal domain. One of the important aims of video compression is removal of the temporal redundancy in an efficient way. Motion Estimation (ME), which tries to remove the temporal redundancy by finding the best matching block in a reference frame for each block in the present frame, is the principal component of a video encoding system. Of all the components of a video encoder, the ME module consumes the lion's share of overall power. A very simple arithmetic computation is required for ME. However, frequent memory access associated with ME affects the overall speed of operation and the power consumption. The present work has therefore focused on design and development of several fast ME architectures characterized by high processing speed, low power, and low area making them suitable for portable video application devices that are typically operated by battery power and involve real time operation. VLSI architecture has been developed for Fast Three Step Search (FTSS) algorithm that is used in video conferencing applications. An intelligent data arrangement has been used in this design to reduce the power consumption and to achieve a high speed of operation. Parallel VLSI architectures for Successive Elimination algorithm (SEA) have also been developed. The architecture proposed for SEA requires nearly 60 % less time with same power requirement and accuracy, but somewhat more area while being compared to an architecture meant for realizing full search algorithm. Moreover, the present work has conceived fast ME by combining One Bit Transformation (1BT) for fixed block size and single reference frame. Fast 1BT based ME architectures for variable block size and single reference frame and multiple reference frames have also been developed. The scope of the present work also includes fast ME algorithms based on the pixel truncation. An appropriate architecture has also been developed for implementing the proposed ME algorithm. In the present work, all the proposed architectures have been synthesized and analyzed for power and maximum operating frequencies in FPGA as well as ASIC platforms.

Nowadays, the consumer looks out for the best possible video quality regardless of his/her location and degree of network support. To realize this however, the

transmitted video must match the receiver's characteristics such as the required bit rate, resolution and frame rate, thus aiming to provide the best quality subject to the limitations of the receiver and the network. Scalable video coding provides an appropriate solution to this type of problem. In recent years, wavelet-based image and video coding systems that utilize a wide range of spatial, temporal and SNR scalability with state-of-the-art coding performance have been developed. An introduction to scalable video coding based in-band motion compensated temporal filtering (IB-MCTF) has been provided towards the end of this book.

Keywords Video compression · Fast three step search · Successive elimination · One-bit transformation · Pixel truncation · Parallel VLSI architecture · Scalable video coding

Chapter 1
Introduction

Several image frames combine in a sequence to constitute a video signal. Frame rate of a video processing system is stated to be the number of frames sent out or received per second (fps). As far as a day-to-day consumer application like mobile video communication is concerned, frame rate of 30 fps is considered adequate. However, the rate can vary from as low as 10–15 fps for videoconferencing application to 60 fps for a typical high-end High Definition Television (HDTV) transmission. Storage and transmission of the enormous volume of data that is required for high quality video processing proves to be a challenge for the system designers. At the same time, one cannot overlook the considerable similarity existing between the contents of successive frames, considering the fact that there exists very short time difference (varying between 1/10 and 1/60 s, based on the frame rate) between any two consecutive frames of a video sequence. Removal of this inherent temporal redundancy by employing video compression is the key to development of efficient storage and transmission systems for video information.

Over the last few decades, video compression has been the underlying technology of numerous consumer electronic products including the modern-day smart-phones and tablet computers. Continuing evolution of efficient video processing architecture has enabled development and manufacture of hand held low-area low-power devices including digital camcorders and camera phones.

This introductory chapter first identifies the principal tasks involved in an overall video compression job. Following a brief background on the basic blocks of a typical video codec, the motivation of the present research is stated. Subsequently, challenges faced in undertaking this work are spelt out. Salient contributions of the research work recorded in this book are next enumerated. The chapter ends by drawing an outline of the present book.

1.1 Fundamentals of Video Compression

In this section, we explore the operation of the basic video codec. The major video coding standards released since the early 1990s have been based on the same generic design (or model) of a video CODEC that incorporates a motion estimation and

I. Chakrabarti et al., *Motion Estimation for Video Coding*,
Studies in Computational Intelligence 590, DOI 10.1007/978-3-319-14376-7_1

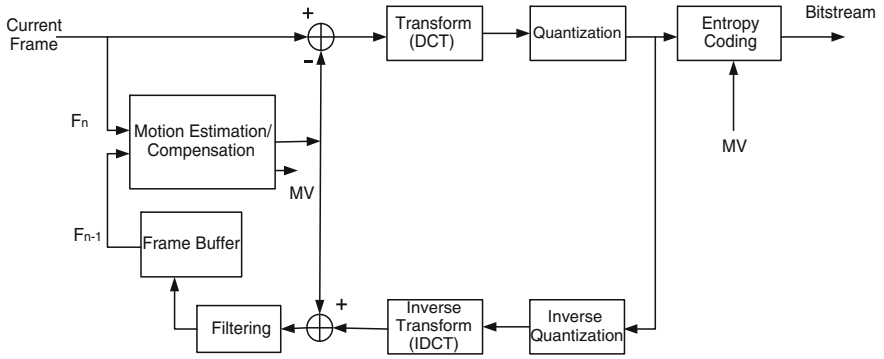


Fig. 1.1 Block Diagram for a generic video encoder

compensation front end (sometimes described as Differential Pulse Code Modulation (DPCM)), a transform stage and an entropy encoder. The model is often described as a hybrid DPCM/DCT CODEC [1, 2]. Any CODEC that is compatible with H.261/263/264/265 or MPEGs (1/2/4) has to implement a similar set of basic coding and decoding functions (although there are many differences of detail between the standards and their actual implementations). Figure 1.1 shows the structure of an encoder for video compression (generic DPCM/DCT hybrid encoder). In an encoder, video frame n(F_n) is processed to produce a coded (compressed) bitstream while in a decoder, the compressed bitstream is decoded to produce a reconstructed video frame $\hat{F}_n$, not usually identical to the source frame. Functionality of each block is explained in the following subsections.

1.1.1 Transform Block

The first step in Discrete Cosine Transform (DCT) based image/video coding is to divide the image into small blocks, usually of size 8×8 pixels. Then, DCT operation is performed on each block to convert each pixel value into frequency domain. It takes 64 input values and yields 64 frequency domain coefficients. This transform is fully reversible; the original block can be reconstructed by applying an Inverse DCT (IDCT). DCT not only converts the pixels into the corresponding frequency values such that the lower frequencies appear at the top-left side of the block, while the higher frequencies appear at the bottom right. As the human eye is sensitive to only low frequencies, subsequently steps tend to discard the high frequency values to achieve compression. Hence, DCT helps separate more perceptible information from less perceptible information. DCT converts a block of image pixels into a block of transform coefficients of the same dimension [3]. These DCT coefficients represent the original pixels values in the frequency domain. Any gray-scale 8×8 pixel block can be fully represented by a weighted sum of 64 DCT basis functions where the weights are just

the corresponding DCT coefficients [1, 2]. The two-dimensional DCT transform of an N × N pixel block is described in Eq. (1.1), where $f(j,k)$ is the pixel value at the position (j,k) and $F(u,v)$ is the transform coefficient at the position (u,v).

$$F(u,v) = \frac{2}{N} C(u)C(v) \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} f(j,k) \cos\left[\frac{(2j+1)u\pi}{2N}\right] \cos\left[\frac{(2k+1)v\pi}{2N}\right] \tag{1.1}$$

Corresponding Inverse DCT is given by

$$f(j,k) = \frac{2}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C(u)C(v)F(u,v) \cos\left[\frac{(2j+1)u\pi}{2N}\right] \cos\left[\frac{(2k+1)v\pi}{2N}\right] \tag{1.2}$$

where

$$C(w) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } w = 0; \\ 1 & \text{for } w = 1, 2, 3, \ldots, n-1; \end{cases}$$

Using formulae (1.1) and (1.2), the forward and inverse transforms require a large number of floating point computations. Simple calculation shows that a total of $64 \times 64 = 4{,}096$ computations are needed to transform a block of 8×8 pixels.

It may be noted that the forward and inverse DCT transforms are separable, implying that the two dimensional transform coefficients can be obtained by applying a one-dimensional transform first along the horizontal direction and then along the vertical direction separately. This can reduce the number of computations required for each 8×8 block from 4,096 to 1,024.

1.1.2 Quantization

For a typical block in a photographic image, most of the high-frequency DCT coefficients will be nearly zero. On an average, the DC coefficient and the other low-frequency coefficients often have relatively large amplitudes. This is because in an image with smooth natural scene, most blocks tend to contain little high-frequency contents; in general, only a few of the DCT coefficients have significant values [1, 2]. The DCT coefficients are quantized so that the near-zero coefficients are set to zero and the remaining coefficients are represented with reduced precision. To quantize each coefficient, it is divided by the quantizer step size and the result is rounded to the nearest integer. Therefore, larger quantizer step sizes mean coarser quantization. Although this results in information loss, compression is achieved since most of the coefficient values in each block now are zero. Coarser quantization (i.e., larger quantizer step size) gives higher compression and poorer decoded image quality.

1.1.3 Entropy Coding

After quantization, nonzero coefficients are further encoded using an entropy coder such as Huffman coder. In Huffman coding [4] (and in other entropy coding schemes) [1, 2], the more frequent values are represented with shorter codes and the less frequent values with longer codes. Zero coefficients can be efficiently encoded using run-length encoding. Instead of transmitting all the zero values one by one, run length coding simply transmits the total number of the current run of zeros. The result is a compressed representation of the original image. To decode the image, the reverse procedure is carried out. First, the variable-length codes (entropy codes) are decoded to get back the quantized coefficients. These are then multiplied by the appropriate quantizer step size to obtain an approximation to the original DCT coefficients. These coefficients are put through the inverse DCT to get back the pixel values in the spatial domain. These decoded pixel values will not be identical to the original image pixels since a certain amount of information is lost during quantization. A lossy DCT CODEC produces characteristic distortions due to the quantization process. These include "blocking" artifacts [5], where the block structure used by the encoder becomes apparent in the decoded image, and "mosquito noise" [5], where lines and edges in the image are surrounded by fine lines. DCT-based image coding systems can provide compression ratios ranging between 10 and 20 while maintaining a reasonably good image quality. The actual efficiency depends to some extent on the image content. As images with considerable detail contain many nonzero high-frequency DCT coefficients, they need to be coded at higher rates than images with less detail. Compression can be improved by increasing the quantization step size. In general, higher compression is obtained at the expense of poorer decoded image quality.

1.1.4 Motion Estimation and Compensation

Motion Estimation (ME) and Motion Compensation (MC) play an important role in video coding system. Motion estimation has been adopted in many video coding standards like MPEG-X series and H.26X series [6]. In motion estimation and motion compensation, the previous or the future frame is used as the reference frame to predict the current frame [7]. Both the frames are divided into blocks of fixed sizes, usually 16×16 pixels known as the Macroblocks (MBs). For each MB in the current frame, one estimates its motion by searching for the best matched MB (within a search range) in a previously available reference frame.

The displacement between the MB in the current frame and the best matched one in the reference frame is known as the Motion Vector (MV). It is only the difference between the two MBs and the displacement used to describe the position of the reference MB, which needs to be coded. The amount of coded data is therefore much less than the original, and a high compression ratio can be achieved. However, modern coding standards like H.264 allow ME for blocks of variable sizes to achieve a better prediction for objects with complex motion [1, 2].

As mentioned previously, the purpose of motion estimation is to search for the most similar reference block for a given block in the current frame and to find a MV for the same. This process is repeated for all the blocks in the current frame and a separate motion vector is found for all the blocks. On the other hand, the process of motion compensation is concerned with reconstructing the current frame from the MVs which have been obtained from the motion estimator.

Motion estimation happens to be the most computationally expensive and resource hungry operation in the entire video compression process. Hence, this field has seen the highest activity and research interest in the past two decades.

Many fundamental block matching algorithms [7–21] are available in the literature from the mid-1980s to the recent years. Efficient motion estimation algorithms and architectures are therefore indeed necessary for the design of an efficient video encoder. The following chapters of this book present an in-depth of discussion on some important block motion estimation algorithms and their hardware implementation.

1.2 Motivation

Modern video coding standards have made a substantial amount of improvement possible in the coding efficiency while compared to the previous standards. The required coding gain however entails hardware systems of higher complexity. Motion Estimation (ME) has been identified as the main source of power consumption in the video encoding systems. Although ME based on Block Matching Algorithms (BMAs) involves simple and straightforward arithmetic computations, it requires a huge amount of memory access which in turn calls for considerable power consumption and it may also affect the overall speed of operation [8]. However, there are not many implementations, which ensure low power consumption and high speed of operation at the same time, that are found in the literature. Design of an architecture for ME, which can take care of the high memory bandwidth requirement, is thus crucial for an efficient video encoding system. Therefore, a major objective of the present work has been to design hardware structures for ME with low power requirements. Several techniques have been adopted in this work to design ME architectures endowed with the desirable features of low area and low power consumption, and high speed of operation to meet the real time requirements for the modern video coding/processing applications.

With rapid evolution of digital video technology and the continuous improvement taking place in communication infrastructure, the consumer demands the best possible video quality wherever they are and whatever their network support is. For this purpose, the transmitted video must match the receiver's characteristics such as the required bit rate, resolution and frame rate, thus aiming to provide the best quality subject the limitations of the receiver and the network. Besides, the same link is often used to transmit to either low-end devices such as small cell phones, or to high-performance devices like HDTV workstations. Based on this observation, it is

evident that such heterogeneous networks pose a great problem for traditional video encoders which do not allow for on-the-fly video streaming adaptation.

To circumvent this drawback, the concept of scalability for video coding has been proposed as an emergent solution for supporting, in a given network, endpoints with distinct video processing capabilities. Scalable Video Coding (SVC) must support more than one spatial, temporal and quality layers. This demands a more advanced codec structure of SVC in comparison to that of the conventional hybrid video coding structure. This book provides an introduction to SVC in Chap. 7. It has many potential applications ranging from High Definition Digital Video Disc (HD-DVD) to Digital Video Broadcasting for Hand held terminals (DVB-H) with small screens. However, with the SVC coding performance comes the overhead of high computation complexity. Employing a general purpose processor to deal with such high complexity will necessitate considerable increase in power budget and overall cost. Dedicated hardware (based on design of VLSI Architecture) can only ensure acceptable performance of high resolution video processing. So for real-time applications, need is felt for a dedicated hardware accelerator system. This is the motivation behind the work on designing architectures for SVC based on In-Band Motion Compensated Temporal Filtering (IB-MCTF) to address video communication over heterogeneous networks that involves video data communication over variable rate and bandwidth conditions.

1.3 Challenges Encountered

Due to computational regularity, motion estimation based on full search is generally preferred for Very Large Scale Integration (VLSI) implementation. However, the computational complexity for the full search algorithm is very huge and therefore many fast search algorithms have been proposed.

- Although the computational time of these fast search algorithms is much smaller than full search algorithm, the search data flow for these fast search algorithms is irregular when one becomes involved with their hardware realization. This makes the memory access mechanism for VLSI implementation of fast search algorithms more complex than that for full search algorithm.
- Unlike full search algorithm, the processing order of these fast search algorithms is not predefined but dynamic, which makes the controlling unit much more complicated than that of the full search algorithm.

In recent years, wavelet-based image and video coding systems that utilize a wide range of spatial, temporal and SNR scalability with state-of-the-art coding performance have been proposed in the literature [22–26].

- To serve a broad range of data rates (that vary from a few Kbit/s to several Mbit/s) on heterogeneous networks or on a wide variety of terminals with different characteristics, fine-granular spatio-temporal and SNR scalability becomes necessary.

- Selecting trade-offs among these three dimensions (spatial/temporal/quality) becomes inevitable in order to support a high degree of content variation with high quality. Motion-compensated wavelet video coding schemes can provide full scalability with fine granularity over a large range of bitrates.

These factors pose considerable challenges for designing VLSI architectures for fast search ME algorithms and SVC.

Many techniques have been applied to reduce the overall power consumption of the ME modules by reducing the required memory bandwidth. This is achieved by reducing the redundant data access for different search locations. Additionally, highly parallel architectures have been developed to increase the overall speed of operation for the ME process. Therefore, a major part of this book beginning from Chap. 3 and ending in Chap. 6 has focused on the design and development of fast ME architectures characterized by high processing speed, low power, and low area, which make them suitable for portable video application devices that are typically operated by battery power and involve real time operation. Chapter 7 gives an introduction to Scalable Video Coding based on spatial domain motion compensated temporal filtering (SD-MCTF) and in-band motion compensated temporal filtering (IB-MCTF).

1.4 Contributions of the Present Research

Keeping the challenges stated above in view, the main contributions of the work embodied in this book have been the following:

- To optimize the design of the proposed architectures to meet the requirement for low-power consumption.
- To design and implement efficient VLSI architecture for Fast Three Step search (FTSS) motion estimation algorithm [27]. FTSS determines the direction of the current motion vector from the previous motion vector and reduces the computation for checking the candidate motion vector.
- To develop a parallel VLSI architecture for successive elimination algorithm (SEA) [28]. By using SEA, motion vector for each reference block in the current frame can be found with much less computational load than what would be required if one executes the exhaustive search algorithm.
- To design and develop a high performance ME architectures based upon combination of diamond search algorithm and one-bit transformation supporting blocks of variable sizes and multiple reference frames [29].
- To design and develop an efficient architecture to reduce the computational complexity and memory access for variable block size ME based on pixel truncation [30].
- To optimize the performance of each ME architecture to meet the requirements for modern video coding standards.
- To perform comparison with existing work in order to establish the acceptability of the proposed architectures.
- To give an introduction to scalable video coding based on in-band motion compensated temporal filtering (IB-MCTF) through lifting based DWT.

1.5 Organization of the Book

This book primarily focuses on low-power VLSI implementation of ME architectures and efficient data reuse technique along with other techniques that have been used to make a high performance ME architecture. Finally the concept of scalable video coding based on in-band motion compensated temporal filtering has also been presented.

In this chapter gives a brief introduction to the concept of video compression and ME, different ME architectures and explains the motivation of the present work. It also identifies the major challenges faced and lists the principle contributions made in course of this research.

Chapter 2 provides the background of ME and different fast search techniques for motion estimation and brief survey on literature related to the scalable video coding.

Chapter 3 explains the design and development of VLSI architecture for realizing Fast Three Step Search algorithm (FTSS).

Chapter 4 describes design and implementation of VLSI architecture implementation of Successive Elimination Algorithm (SEA).

Chapter 5 provides the details on architectural implementation of fast ME based on a combination of Diamond Search and 1-bit transformation.

A two stage fast algorithm for Variable Block Size Motion Estimation (VBSME) based on pixel truncation has been proposed in this work. A suitable low-power architecture for implementing the proposed ME algorithm has been described in Chap. 6.

Chapter 7 briefly explains the work done so far in IB-MCTF which is an important branch of the broad field of scalable video coding.

Finally Chap. 8 presents a few suggestions for extensions of the present work.

To give the reader a better feel for the approach adopted in the present work, some important Matlab and Verilog programs are given in Appendix A and Appendix B respectively.

References

1. Richardson, I.E.G.: H.264 and MPEG-4 Video Compression Video Coding for Next Generation Multimedia. Wiley, West Sussex (2003)
2. Tekalp, A.M.: Digital Video Processing. Prentice-Hall, Upper Saddle River (1995)
3. Watson, A.B.: Image compression using the discrete cosine transform. Math. J. **4**(1), 81–88 (1994)
4. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proc. IRE **40**(9), 1098–1101 (1952). doi:10.1109/JRPROC.1952.273898
5. Meesters, L., Martens, J.-B.: Blockiness in JPEG-coded images, HVEI IV, vol. 3644. SPIE, San Jose (1999)
6. Puria, A., Chen, X., Luthra, A.: Video coding using the H.264/MPEG-4 AVC compression standard. Signal Process.: Image Commun. **19**, 793–849 (2004)
7. Gharavi, H., Mills, M.: Blockmatching motion estimation algorithms-new results. IEEE Trans. Circuits Syst. **37**(5), 649–665 (1990)

8. Kawahito, S., Handoko, D., Tadokoro, Y., Matsuzawa, A.: Low power motion vector estimation using iterative search block-matching methods and a high-speed non-destructive CMOS sensor. IEEE Trans. Circuits Syst. Video Technol. **12**(12), 1084–1092 (2002)
9. Li, R., Zeng, B., Liou, M.L.: A new three-step search algorithm for block motion estimation. IEEE Trans. Circuits Syst. Video Technol. **4**(4), 438–442 (1994)
10. Lu, J., Liou, M.L.: A simple and efficient search algorithm for block-matching motion estimation. IEEE Trans. Circuits Syst. Video Technol. **7**(2), 429–433 (1997)
11. Po, L.-M., Ma, W.-C.: A novel four step search algorithm for fast block motion estimation. IEEE Trans. Circuits Syst. Video Technol. **6**(3), 313–317 (1996)
12. Li, W., Salari, E.: Successive elimination algorithm for motion estimation. IEEE Trans. Image Process. **4**(1), 105–107 (1995)
13. Zhu, S., Ma, K.K.: A new diamond search algorithm for fast block-matching motion estimation. IEEE Trans. Image Process. **9**(2), 287–290 (2000)
14. Nie, Y., Ma, K.K.: Adaptive rood pattern search for fast block-matching motion estimation. IEEE Trans. Image Process. **11**(12), 1442–1448 (2002)
15. Cheung, C.H., Po, L.M.: A novel cross-diamond search algorithm for fast block motion estimation. IEEE Trans. Circuits Syst. Video Technol. **12**(12), 1168–1177 (2002)
16. Ghanbari, M.: The cross search algorithm for motion estimation. IEEE Trans. Commun. **38**(7), 950–953 (1990)
17. Chen, L.G., Chen, W.T., Jehng, Y.S., Chiueh, T.D.: An efficient parallel motion estimation algorithm for digital image processing. IEEE Trans. Circuits Syst. Video Technol. **1**(4), 378–385 (1991)
18. Tourapis, A.M., Au, O.C., Liu, M.L.: Highly efficient predictive zonal algorithms for fast block-matching motion estimation. IEEE Trans. Circuits Syst. Video Technol. **12**(10), 934–947 (2002)
19. Natarajan, B., Bhaskaran, V., Konstantinides, K.: Low-complexity block-based motion estimation via one-bit transforms. IEEE Trans. Circuits Syst. Video Technol. **7**(3), 702–706 (1997)
20. Liu, B., Zaccarin, A.: New fast algorithms for the estimation of block motion vectors. IEEE Trans. Circuits Syst. Video Technol. **3**(2), 148–157 (1993)
21. Hsieh, C.H., Lu, P.C., Shyn, J.S., Lu, E.H.: Motion estimation algorithm using interblock correlation. IEE Electron. Lett. **26**(5), 276–277 (1990)
22. Andreopoulos, Y., van der Schaar, M., Munteanu, A., Barbarien, J., Schelkens, P., Cornelis, J.: Complete-to-overcomplete discrete wavelet transforms for scalable video coding with MCTF. Proc. SPIE/IEEE Visual Commun. Image Process. **5150**, 719–731 (2003)
23. Ohm, J.-R.: Advances in scalable video coding. Proc. IEEE, Invit. Pap. **93**, 42–56 (2005)
24. Wang, B., Loo, K.K., Yip, P.Y., Siyau, M.F.: A simplified scalable wavelet video codec with MCTF Structure. In: International Conference on Digital Telecommunications (ICDT'06), pp. 29–31 August 2006. doi:10.1109/ICDT.2006.11
25. Andreopoulos, Y., van der Schaar, M., Munteanu, A., Barbarien, J., Schelkens, P., Cornelis, J.: Fully-scalable wavelet video coding using in-band motion-compensated temporal filtering. In: Proceedings IEEE International Conference Acoustics, Speech and Signal Processing, pp. III-417–III-420 (2003)
26. Park, H.-W., Kim, H.-S.: Motion estimation using low-band-shift method for wavelet-based moving-picture coding. IEEE Trans. Image Process. **9**(4), 577–587 (2000)
27. Srinivasarao, B.K.N., Chakrabarti, I.: A parallel architectural implementation of the fast three step search algorithm for block motion estimation. In: Proceedings of 5th International Multi-Conference on Systems, Signals and Devices (SSD-08), Amman, Jordan, 20–22 July 2008, pp. 1–6. doi:10.1109/SSD.2008.4632849

28. Srinivasarao, B.K.N., Chakrabarti, I.: A parallel architecture for successive elimination block matching algorithm. In: Proceedings TENCON-2008 (IEEE Region 10 Conference), pp. 1–6. Hyderabad, India, 19–21 November 2008
29. Chatterjee, S.K., Chakrabarti, I.: Low power VLSI architectures for one bit transformation based fast motion estimation. IEEE Trans. Consum. Electron. **56**(4), 2652–2660 (2010)
30. Chatterjee, S.K., Chakrabarti, I.: Power efficient motion estimation algorithm and architecture based on pixel truncation. IEEE Trans. Consum. Electron. **57**(4), 1782–1790 (2011)

Chapter 2
Background and Literature Survey

This chapter begins with an overview of block matching algorithm (BMA) approach to motion estimation which is preferred for its simplicity and straightforward circuit implementation. Many block matching algorithms are briefly introduced and also a brief survey of different motion estimation architectures are presented.

2.1 Block Matching Algorithm

Mainly, there are two different techniques of ME, namely Pel-Recursive Algorithm (PRA) and Block Matching Algorithm (BMA). In PRAs, there is an iterative refining of ME for individual pixels by gradient methods [1]. On the other hand, BMAs assume that all the pixels within a block have the same motion activity [2]. In BMAs, motion is estimated on the basis of rectangular blocks and one Motion Vector (MV) is produced for each block. Compared to BMAs, PRAs involve more computational complexity and less regularity, and so are difficult to realize in hardware. In general, BMAs are more suitable for a simple hardware realization because of their regularity and simplicity [3]. Also, BMA is adopted in all video coding standards because of its performance [4]. In the process of BMA, one is required to find a MB in the reference frame within a given search area, that is most similar to the MB in the current frame (current MB). Due to a given search range a window like structure is formed in the reference frame, which is known as the Search Window (SW). For a search range of $[-p, +p]$ and for a MB of size $N \times N$, the spatial relationship between the current MB and the SW is shown in Fig. 2.1. The matching criterion of the BMA has a direct impact on coding efficiency and computational complexity. Many matching criteria have been proposed in literature e.g., mean squared error, Sum of Absolute Differences (SAD), pel difference classification etc. [5]. Among the various proposed matching criteria, SAD calculation requires only a few simple computational steps, and thus is the most preferred one for VLSI

I. Chakrabarti et al., *Motion Estimation for Video Coding*,
Studies in Computational Intelligence 590, DOI 10.1007/978-3-319-14376-7_2

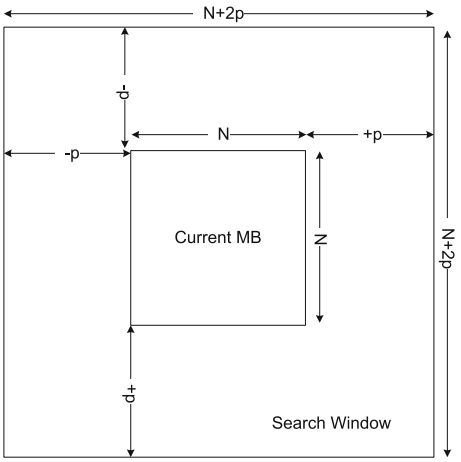


Fig. 2.1 The process of motion estimation by block matching algorithm

implementation. The evaluation of SAD for a given location (m, n) within the SW is done as:

$$SAD(m,n) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |cur(i,j) - ref(i+m, j+n)| \tag{2.1}$$

where $-\text{p} \leq \text{m, n} \leq +\text{p}$. Also, $cur(i, j)$ is the current MB of size N × N at the coordinate location (i, j), while $ref(i + m, j + n)$ is the reference block within the SW at the coordinate location $(i + m, j + n)$ and p is the search range in both the directions. The term $|cur(i, j) - ref(i + m, j + n)|$ is known as the distortion which is the absolute difference in intensity between the current pixel cur(i, j) and the reference pixel $ref(i + m, j + n)$ [4]. The expression SAD(m, n) yields the summation of all the distortions for the current MB at the search location (m, n). The search candidate, which has the smallest SAD, is selected as the best matching reference MB, and the associated location (m, n) is the MV of this current MB. In the following subsections, ME algorithms are broadly classified into two categories, namely full search algorithm and fast search algorithms based on the provided quality and the search process.

2.1.1 Full Search Block Matching Algorithm

In full search BMA, all search candidates within the search window are evaluated, and the search candidate with the smallest SAD is selected as the best matched search candidate. The final MV is obtained from the location of the best matched search candidate. The algorithm for full search based ME is shown in Algorithm 2.1. Since all the search candidates are examined in full search, ME based on the full

search provides the optimum solution. Although full search yields optimum results, it requires a huge amount of computation.

Algorithm 2.1 Full Search Block Matching Algorithm

1: $SADmin = MAXVALUE$;
2: $MV = (0, 0)$;
3: **for** m = -p to +p **do**
4: **for** n = -p to +p **do**
5: SAD(m,n) = 0;
6: **for** i = 0 to N-1 **do**
7: **for** j = 0 to N-1 **do**
8: $SAD(m, n) = SAD(m, n) + |cur(i, j) - ref(i + m, j + n)|$
9: **end for**
10: **end for**
11: **if** SAD(m,n) < SADmin **then**
12: $SADmin = SAD(m, n)$;
13: $MV = (m, n)$;
14: **end if**
15: **end for**
16: **end for**

For example, the computational complexity required to perform ME in real time for a video sequence in Common Intermediate Format (CIF) (352 × 288 @ 30 fps) and for a search range of size [−16, 15] is 9.3 Giga Operations per Second (GOPS). If the frame size becomes DVD format (720 × 480 @ 30 fps) and the searching range is increased to [−32, 31], the required computational complexity also increases to 127 GOPS. This extremely large computational complexity for full search based ME has motivated the development of many fast search algorithms. In the next subsections, several fast search algorithms will be discussed.

2.1.2 Fast Search Algorithms for Block Matching Algorithm

In order to reduce the huge computational requirement for motion estimation based on full search algorithm, a large number of fast but sub-optimal BMAs can be found in the literature [6–11]. These algorithms reduce the computational time as well as the hardware overhead to a considerable extent. However, the major drawback associated with these fast search algorithms is that very often they may be trapped in some local minima and thereby produce suboptimal results. Fast search algorithms can be broadly classified into three categories, namely (i) reduction in the number of search candidates [11–15] (ii) exploiting different matching criteria instead of the classical SAD [16–18] , and (iii) predictive search [19–22] based on their characteristics.

In the following subsection, a brief introduction to these categories and some typical examples are presented.

2.1.2.1 Reduction in the Number of Search Candidates

These algorithms are based on the assumption that the distortion monotonically decreases as the search candidate approaches the optimal one. That is, even if all the search candidates are not matched, the optimal search candidate can be obtained by following the search candidate with the smallest distortion. This category accounts for the majority of fast search algorithms, and there are many algorithms available in literature, such as three step search [12], Successive Elimination, cross search [13], new-three-step search [6], four step search [12], unrestricted center-biased diamond search [23], diamond search [9], and so on.

Figure 2.2 depicts the search process for the Three Step Search (TSS) [12]. This algorithm was introduced by Koga et al. [12]. It became very popular because of its simplicity. It searches for the best motion vectors in a coarse-to-fine search pattern. In the first step, an initial step size is fixed. Eight blocks at a distance of the step size from the center (around the center block) are picked for comparison. In the next step, the center is moved to the point giving the minimum distortion with the step size halved. This is repeated till the step size becomes smaller than 1. One problem that occurs with TSS is that, it uses a uniformly allocated checking point pattern in the first step, which becomes inefficient for small motion estimation.

In Successive Elimination algorithm, motion vector for each reference block in the current frame can be find with much less computational load than exhaustive search algorithm by using some mathematical properties, which is discussed in Chap. 3.

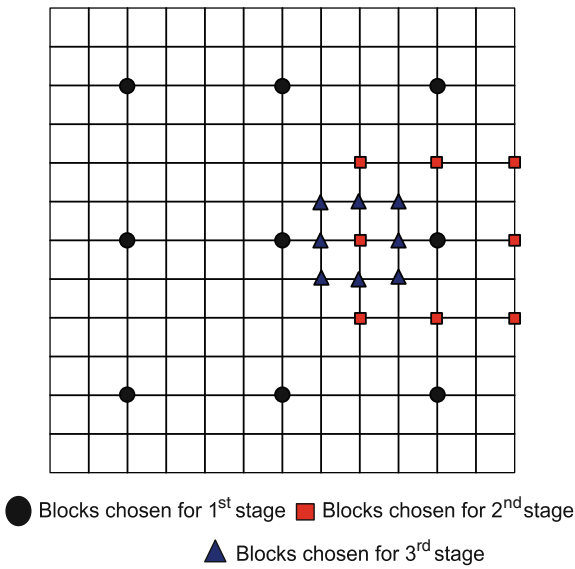


Fig. 2.2 The search process for three step search

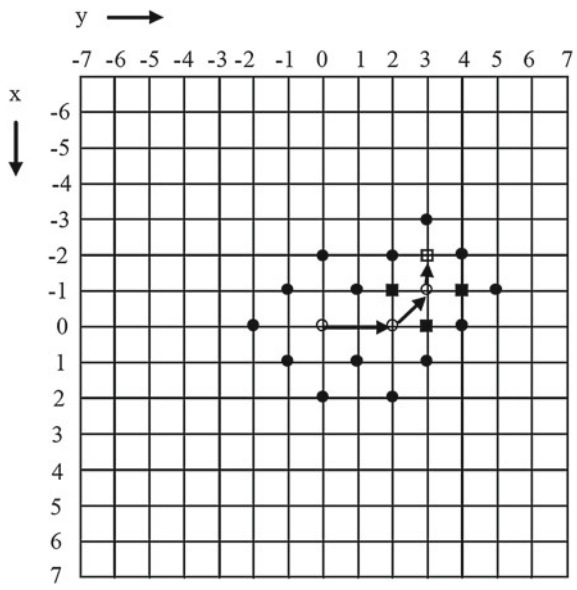


Fig. 2.3 The search procedure for diamond search algorithm

Diamond Search (DS) [9] is another typical fast search algorithm and is shown in Fig. 2.3 DS has two search steps namely, large diamond and small diamond. In the searching procedure, the large diamond step is applied first. DS continues in the large diamond step until the search candidate at the center has the smallest distortion among the nine candidates of the large diamond. Next, the small diamond is used to refine the searching result of the large diamond. Figure 2.3 portrays an example of DS algorithm. The arrow is the direction toward which the large diamond moves, and after the searching result of the large diamond converges, the small diamond is adopted to refine the searching result in the last step.

2.1.2.2 Simplification of Matching Criteria

The matching criterion of the block matching ME method has a direct impact on the coding efficiency and the computational complexity. Many matching criteria have been proposed in literature e.g., mean square error, SAD, pel difference classification etc. [5]. Whatever may be the matching criterion, evaluation of the matching criterion on pixels with 8 bits/pixel representation requires a huge amount of computation. The computational load can be reduced to a great extent by representing the pixels with a reduced number of bits. This method is known as pixel truncation. As proposed in [24], the number of bits in each pixel is truncated to achieve the reduction in computation. For example, if the number of bits in each pixel is truncated from eight bits to five bits, then the required computational load is only 5/8 of the original. Moreover, not only does pixel truncation serve to reduce the computational complexity, it also saves the hardware cost and the power consumed by ME hardware.

This is because a subtractor with less bit width can be used instead of that with eight bits. In the majority of video sequences, any pixel can be truncated to only six or five bits without much degradation in the quality.

It has been shown in [25] that for motion estimation based on pixel truncation, optimum results may be obtained by using Difference Pixel Count (DPC) as the matching criteria instead of the conventional SAD. For a block of size N × N and for a search range [−p, p − 1], the DPC at any location (m, n) can be found as [25]:

$$DPC(m,n) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \bar{\delta}[\hat{C}(i,j), \hat{R}(i+m, j+n)] \tag{2.2}$$

Here, $\hat{C}(i, j)$ and $\hat{R}(i + m, j + n)$ represent bit truncated values for the pixels from the CB and the SW respectively. In Eq. 2.2, $\bar{\delta}(x, y)$ represents the standard delta function, for which $\bar{\delta}(x, y) = 0$ if (x = y); else its value is 1.

In another method, as proposed in [16], an image frame with 8 bits/pixel representation is first converted into a binary frame with 1 bit/pixel representation. Motion estimation is then carried out on these binary image frames. Boolean exclusive OR (XOR) operation is used to find the Number of Non-Matching Points (NNMP), which is used as the matching criterion in place of the conventional SAD. The NNMP at any point (m, n) for a MB of size N × N is found as:

$$NNMP(m,n) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} B^t(i,j) \oplus B^{t-1}(i+m, j+n)] \tag{2.3}$$

where, $-s \leq m, n \leq s$

Here, s is the maximum search range and ⊕ denotes the XOR operation. Also, B^t and B^{t-1} represent the current and the reference 1BT frames respectively.

2.1.2.3 Predictive Search

The main problem associated with the fast search algorithms is that they are usually trapped into a local minimum. In order to avoid this condition, predictive search is developed and combined with other fast search algorithms. The concept of predictive search is based on the assumption that the Motion Vectors (MVs) of neighboring MBs are correlated and similar, so that they can be used to predict the MV of the current MB. Besides the spatial information, the temporal information also can be used in the process of prediction because of the motion continuity in the temporal direction. Therefore, the motion information of neighboring blocks in the spatial or temporal space is used to serve as the initial search candidate of fast search algorithms instead of the original point.

As proposed in [15], the initial search candidate can be the MVs of the blocks on the top, left, and top-right, their median, zero MV, the MV of the collocated block

in the previous frame, and the accelerated motion vector of the collocated block in the previous two frames. By this way, the searching range can be reduced and constrained, so that not only the computational complexity but also the bit-rate of MV can be reduced. The Fast three step search algorithm (FTSS) [26] makes use of the directional information from adjacent previous motion vector and unimodal error surface assumption (UESA). It determines the direction of the current motion vector from the previous motion vector and reduces the computation for checking the candidate motion vector using the UESA. The UESA means that the error increases monotonically in getting away from the global minimum [27].

2.1.3 Motion Estimation Architectures

In order to achieve real time computation of Motion Estimation (ME), it is required that the ME hardware should be fast and at the same time should consume low power. Many ME architectures have been proposed in the last few years. In general, ME hardware can be broadly divided into two parts, the Processing Element (PE) array and the on-chip memory, as shown in Fig. 2.4.

PE array is the major operational core and responsible for the computation of SAD as given by Eq. 2.1. Although motion estimation involves simpler arithmetic computations, it involves a huge amount of memory access which involves considerable power consumption and also affects the overall speed of operation [24]. The required data are loaded through a global data bus. Moreover, in order to reduce the required memory bandwidth, some of the data are stored in the on-chip memory for data re-use. For each Macroblock (MB), the PE array gets the required data from both the global/system bus and the on-chip memory, and computes the corresponding SADs. At the same time, the data in the on-chip memory are updated for data re-use of the next MB or the search candidate. Depending on different ME algorithms and characteristics of PE array, the ME architectures can be broadly classified into three types: inter-level, intra-level, and tree-based architectures. While all inter-level and intra-level architectures have been used mostly for implementing full search algorithm,

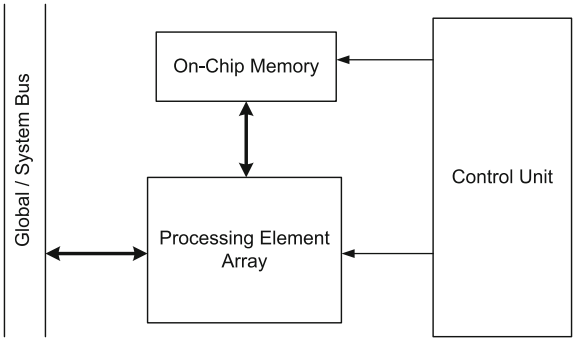


Fig. 2.4 The block diagram of a typical ME architecture

tree-based architectures have been used for realizing fast search algorithms. In the following subsections, a brief survey of ME architectures are presented.

2.1.3.1 Motion Estimation Architectures for Full Search Algorithm

The full search algorithm can provide the best quality among various ME algorithms, but involves a huge amount of computation. Although the computational requirement for full search is large, it is preferred for VLSI implementation because of its simple operations and regular data flow compared to the fast search algorithms. Many different types of architectures have been proposed for the full search algorithm. The inter-level and intralevel architectures are the most commonly used. The first VLSI implementation of motion estimator was due to Yang et al. [28], who implemented the ME architecture for full search algorithm. This architecture was based on inter-level ME architecture. The PE in inter-level architectures is responsible for the computation of one search location. One PE computes the differences of all the pixels in the current block and accumulates the SAD pixel by pixel. The partial SAD is stored in each PE, until the SAD calculation of one search location is finished. A comparator is responsible for selecting the minimum SAD among all the generated SADs.

A detailed systolic mapping procedure to derive full search BMA architectures was proposed by Komarek and Pirsch [29]. This architecture was based on intra-level architecture. In an intra-level architecture, current pixels are stored in the corresponding PEs, and the reference pixels are propagated from one PE to another. The PE is responsible for calculating the distortion between one specific current pixel and the corresponding reference pixel for all the search candidates. In each PE, the distortion of a current pixel in current MB is computed and added to the partial SAD, which is propagated from the other PEs. After the initial cycles (depending upon the block size), the SADs are generated one by one and the comparator selects the minimum of them.

A large number of architectures have been proposed based on these two basic architectures. For example, the extension of architecture [28] has been proposed in [30], and besides one-dimensional inter-level architectures, two two-dimensional inter-level semisystolic architectures have been proposed in [31, 32]. The architecture [33] is an extension of [29]. The architectures described in [34] and [35] are two other intra-level architectures with large registers for fewer data inputs and memory bandwidth.

2.1.3.2 Motion Estimation Architectures for Fast Search Algorithms

Unlike the full search algorithm, the data flow for fast search algorithms is irregular and the processing order of the search candidates is not predefined. Rather, it depends on the last searching result. Thus, although the computational complexity of fast search algorithms is much smaller than that of the full search algorithm, the irregular data flow required for the fast search algorithms poses considerable challenge for

VLSI implementation. This makes the implementation of fast search algorithms much more difficult than that of the full search algorithm. Jong et al. [36] developed a fully pipelined parallel architecture for three step search BMA. Basically, 9 PEs compute the SAD of nine candidates in each step, and 256 cycles are required in each of the three steps. An intelligent data and memory arrangement are used to utilize the advantage of three step search procedure. Tree-based architectures developed by Jong et al. have the advantages of short latency, support for random access of the search candidates, and absence of pipelining bubbles cycles. A tree-based architecture that can support DS and fast full search algorithms has been proposed in [37]. As shown in Fig. 2.3, there are many duplicated search candidates between two successive steps in DS. After each search step for large diamond pattern, only five or three search candidates need to be calculated, and the others are calculated at the last large diamond pattern. In order to avoid the duplicated search candidates, a ROM-based solution, which uses a ROM to check if the search candidate is required to be computed or not, has been proposed in [37]. As stated in [37], the ROM-based solution can save 24.4 % search candidates in the DS algorithm, and the area overhead is also not significant. This architecture also supports fast full search algorithms.

Several architectures have been proposed for fast search algorithms besides tree-based architectures. For example, Dutta and Wolf [38] have modified the data flow of the 1-D linear array in [28] to support full search, TSS, and the conjugate direction search in the same architecture. A joint algorithm architecture design of a programmable motion estimator chip has been proposed by Lin et al. [39]. Cheng and Hang have proposed architecture based on universal systolic arrays to realize many BMAs [40].

2.2 Scalable Video Coding

Scalable Video Coding (SVC) must support more than one spatial, temporal and quality layer; hence the Codec structure of SVC differs from the conventional hybrid video coding structure of the H.264 video standard. There have been many contributors to the codec structure for an SVC. The first such contribution to the SVC codec structures was by Ohm [41]. Some of his video codec designs were modifications of the conventional hybrid coding [42–46]. Hybrid coding has been prevalent in all the video coding standards since the introduction of motion compensation for video coding. In case of encoding of the hybrid video structure, one frame predicts another, the predicted frame predicts another and this goes on for a GOP. It is quite evident and stated in [47] that prediction error tends to accumulate and the quality of the frames worsens as we move towards the last frame of the GOP. To avoid such a problem Motion Compensated Temporal Filtering has been introduced [41, 48–50].

Application of Motion Compensation (MC) is a key for high compression performance in video coding, but still is often understood to be implicitly coupled with frame prediction schemes. There is indeed no justification for this restriction, as MC can rather be interpreted as a method to align a filtering operation along the

temporal axis with a motion trajectory. In the case of MC prediction, the filters are in principle linear predictive coding (LPC) analysis and synthesis filters, while in cases of transform or wavelet coding, transform basis functions extended over the temporal axis are subject to MC alignment. This is known as motion-compensated temporal filtering. If MCTF is used in combination with a 2-D spatial wavelet transform, this shall be denoted as a 3-D or (depending on the sequence of the spatial and temporal processing) either as a 2-D+t or t+2-D wavelet transform. In case of MCTF as shown in Fig. 2.5, the error frame is used to update the reference frame; hence the error remains within the candidate and reference frames and is not accumulated or propagated to the successive frames.

Andreopoulos et al. [48, 49], introduced Wavelet-Based Scalable Video Coding. Instead of the conventional DCT, DWT was introduced as a suitable Image and Video Transform. Discrete Wavelet Transform is a multi-resolution transform. By using this property, DWT provides for an improved representation of the digital video data in a hierarchical manner. The latter being a useful property which can be extensively used in case of SVC. First codec structure which introduces the property of MCTF was SD-MCTF [48]. SD-MCTF perform the motion compensation in temporal axis on the pixel domain. The residual frames were then spatially decomposed using a suitable Discrete Wavelet Filter. For better coding efficiency [41], IB-MCTF [49] was introduced. Because the motion compensation was performed in the wavelet domain, the problem of shift-variance will be seen. In order to solve this problem, Over-complete DWT [48, 49, 51, 52] has to be performed. The algorithm and complexity models of IB-MCTF were shown in [49, 51]. Conventionally wavelet decomposition was performed by the convolution method. The problem with convolution based DWT was higher memory requirement and greater computational time. To avoid this problem a mathematical approach called Lifting scheme was introduced. Factorization of Discrete Wavelet Filters was done by Daubechies and Sweldens [53] and the results have been used in the implementation of our architecture. Details of over complete DWT (ODWT), lifting based ODWT, SD-MCTF, and IB-MCTF are given in the Chap. 7.

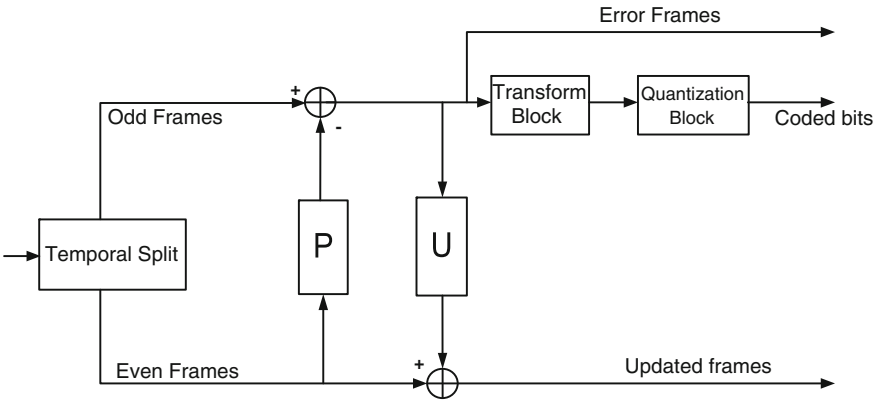


Fig. 2.5 Compensated temporal filtering block with predict and update stage

2.3 Conclusions

This chapter has presented the basic elements of ME algorithms and architectures which will be used in the following chapters. The concepts of motion estimation, block matching algorithm, different ME algorithms, ME architectures, challenges in hardware implementation and advantages of fast search algorithms have been discussed in this chapter. The ME algorithms have been classified into two categories, namely full search and fast search algorithms. Fast search algorithms are further classified based on simplification of matching criteria, reduction of search candidates, and predictive search. The ME architectures on the other hand, are separated into two parts, namely the processing element array and the on-chip memory. In the processing element array, inter-level, intra-level, and tree-based architectures are three basic types of ME architectures and are adopted in many ME architectures and video coding systems.

Scalable video coding based on IB-MCTF has a better PSNR performance than SD-MCTF because there is more degree of freedom of choosing in the ME schemes for the different sub-bands. But the memory requirement and the computational complexity increase as we go down higher levels of spatial and temporal scalability. IB-MCTF is preferred in research oriented fields where the quality of the received video data is more significant. In domains of medical imaging and distant medical applications where the quality of the video is significantly more important than the end-to-end delay, IB-MCTF is better than SD-MCTF.

In the next chapter, an implementation of one of the most popular motion estimation algorithm, namely the Fast Three step search algorithms will be discussed.

References

1. Biemonda, J., Looijengaa, L., Boekeea, D.E., Plompenb, R.H.J.M.: A pel-recursive Wiener-based displacement estimation algorithm. J. Signal Process. **13**(4), 399–412 (1987)
2. Wiegand, T., Sullivan, G.J., Bjontegaard, G., Luthra, A.: Overview of the H.264/AVC video coding standard. IEEE Trans. Circuits Syst. **7**(7), 560–576 (2003)
3. Lu, J., Liou, M.L.: A simple and efficient search algorithm for block-matching motion estimation. IEEE Trans. Circuits Syst. Video Technol. **7**(2), 429–433 (1997)
4. Tekalp, A.M.: Digital Video Processing. Prentice Hall Ltd., New York (1995)
5. Ghanbari, M.: Video Coding, An Introduction to Standard Codecs. The Institute of Electrical Engineers, London (1999). Chaps. 2, 5, 6, 7 and 8.
6. Li, R., Zeng, B., Liou, M.L.: A new three-step search algorithm for block motion estimation. IEEE Trans. Circuits Syst. Video Technol. **4**(4), 438–442 (1994)
7. Po, L.-M., Ma, W.-C.: A novel four step search algorithm for fast block motion estimation. IEEE Trans. Circuits Syst. Video Technol. **6**(3), 313–317 (1996)
8. Li, W., Salari, E.: Successive elimination algorithm for motion estimation. IEEE Trans. Image Process. **4**(1), 105–107 (1995)
9. Zhu, S., Ma, K.K.: A new diamond search algorithm for fast block-matching motion estimation. IEEE Trans. Image Process. **9**(2), 287–290 (2000)
10. Nie, Y., Ma, K.K.: Adaptive rood pattern search for fast block-matching motion estimation. IEEE Trans. Image Process. **11**(12), 1442–1448 (2002)

11. Cheung, C.H., Po, L.M.: A novel cross-diamond search algorithm for fast block motion estimation. IEEE Trans. Circuits Syst. Video Technol. **12**(12), 1168–1177 (2002)
12. Koga, T., Iinuma, K., Hirano, A., Iijima, Y., Ishiguro, T.: Motion compensated interframe coding for video conferencing, Proccedings of Natural Telecommunication Conference, pp. C9.6.1-C9.6.5, (1981)
13. Ghanbari, M.: The cross search algorithm for motion estimation. IEEE Trans. Commun. **38**(7), 950–953 (1990)
14. Chen, L.G., Chen, W.T., Jehng, Y.S., Chiueh, T.D.: An efficient parallel motion estimation algorithm for digital image processing. IEEE Trans. Circuits Syst. Video Technol. **1**(4), 378–385 (1991)
15. Tourapis, A.M., Au, O.C., Liu, M.L.: Highly efficient predictive zonal algorithms for fast block-matching motion estimation. IEEE Trans. Circuits Syst. Video Technol. **12**(10), 934–947 (2002)
16. Natarajan, B., Bhaskaran, V., Konstantinides, K.: Low-complexity block-based motion estimation via one-bit transforms. IEEE Trans. Circuits Syst. Video Technol. **7**(3), 702706 (1997)
17. Liu, B., Zaccarin, A.: New fast algorithms for the estimation of block motion vectors. IEEE Trans. Circuits Syst. Video Technol. **3**(2), 148–157 (1993)
18. Wang, Y., Wang, Y., Kuroda, H.: A globally adaptive pixel decimation algorithm for block-motion Estimation. IEEE Trans. Circuits Syst. Video Technol. **10**(6), 1006–1011 (2000)
19. Hsieh, C.H., Lu, P.C., Shyn, J.S., Lu, E.H.: Motion estimation algorithm using interblock correlation. IEE Electron. Lett. **26**(5), 276–277 (1990)
20. Zafar, S., Zhang, Y.Q., Baras, J.S.: Predictive block matching motion estimation for TV coding-part I: inter-block prediction. IEEE Trans. Broadcast. **37**(3), 97–101 (1991)
21. Zhang, Y.Q., Zafar, S.: Predictive block-matching motion estimation for TV coding-part II: inter-frame prediction. IEEE Trans. Broadcast. **37**(3), 102–105 (1991)
22. Chalidabhongse, J., Kuo, C.C.J.: Fast motion vector estimation using multiresolution-spatio-temporal correlations. IEEE Trans. Circuits Syst. Video Technol. **7**(3), 477–488 (1997)
23. Tham, J.Y., Ranganath, S., Ranganath, M., Kassim, A.A.: A novel unrestricted center biased diamond search algorithm for block motion estimation. IEEE Trans. Circuits Syst. Video Technol. **8**(4), 369–377 (1998)
24. He, Z.L., Tsui, C.Y., Chan, K.K., Liou, M.L.: Low-power VLSI design for motion estimation using adaptive pixel truncation. IEEE Trans. Circuits Syst. Video Technol. **10**(5), 669–678 (2000)
25. Lee, S., Kim, J.M., Chae, S.I.: New motion estimation algorithm using adaptively quantized low bit-resolution image and its VLSI architecture for MPEG2 video encoding. IEEE Trans. Circuits Syst. Video Technol. **8**(6), 734–744 (1998)
26. Kim, J.-N., Choi, T.-S.: A fast three step search algorithm with minimum checking points. In: Proceedings of IEEE Conference Consumer Electronics, pp. 132–133. 2–4 June 1998
27. Jianjua, L., Liou, M.L.: A simple and efficient search algorithm for block-matching motion estimation. IEEE Trans. Circuits Syst. Video Technol. **7**(2), 429–433 (1997)
28. Yang, K.M., Sun, M.T., Wu, L.: A family of VLSI designs for the motion compensation block-matching algorithm. IEEE Trans. Circuits Syst. **36**(2), 1317–1325 (1989)
29. Komarek, T., Pirsch, P.: Array architectures for block matching algorithms. IEEE Trans. Circuits Syst. **36**(2), 1301–1308 (1989)
30. Shen, J.F., Wang, T.C., Chen, L.G.: A novel low-power full search blockmatching motion estimation design for H.263+. IEEE Trans. Circuits Syst. Video Technol. **11**(7), 890–897 (2001)
31. Yeo, H., Hu, Y.H.: A novel modular systolic array architecture for full-search block matching motion estimation. IEEE Trans. Circuits Syst. Video Technol. **5**(5), 407–416 (1995)
32. Lai, Y.K., Chen, L.G.: A data-interlacing architecture with two dimensional datareuse for full-search block-matching algorithm. IEEE Trans. Circuits Syst. Video Technol. **8**(2), 124–127 (1998)
33. Chang, S.F., Hwang, J.H., Jen, C.W.: Scalable array architecture design for full search block matching. IEEE Trans. Circuits Syst. Video Technol. **5**(4), 332–343 (1995)

34. Vos, L.D., Stegherr, M.: Parameterizable VLSI architectures for the full-search block-matching algorithm. IEEE Trans. Circuits Syst. **36**(2), 1309–1316 (1989)
35. Roma, N., Sousa, L.: Efficient and configurable full-search blockmatching processors. IEEE Trans. Circuits Syst. Video Technol. **12**(12), 1160–1167 (2002)
36. Jong, H.-M., Chen, L.-G., Chiueh, T.-D.: Parallel architectures for 3-step hierarchical search block-matching algorithm. IEEE Trans. Circuit Syst. Video Technol. **4**(4), 407–416 (1994)
37. Chao, W.M., Hsu, C.W., Chang, Y.C., Chen, L.G.: A novel motion estimator supporting diamond search and fast full search. In: Proceedings of IEEE International Symposium Circuits Systems (ISCAS02), pp. 492–495 (2002)
38. Dutta, S., Wolf, W.: A flexible parallel architecture adopted to block matching motion estimation algorithms. IEEE Trans. Circuits Syst. Video Technol. **6**(1), 74–86 (1996)
39. Lin, H.D., Anesko, A., Petryna, B.: A 14-GOPS programmable motion estimator for H.26X video coding. IEEE J. Solid-State Circuits **31**(11), 1742–1750 (1996)
40. Cheng, S.C., Hang, H.M.: A comparison of block-matching algorithms mapped to systolic-array implementation. IEEE Trans. Circuits Syst. Video Technol. **7**(5), 741–757 (1997)
41. Ohm, J.-R.: Advances in scalable video coding. Proc. IEEE, Invit. Pap. **93**, 42–56 (2005)
42. Advanced Video Coding for Generic Audiovisual Services, ITU-T Rec. H.264 and ISO/IEC 14496-10 (MPEG-4 AVC), ITU-T and ISO/IEC JTC 1, Version 1: May 2003, Version 2: May 2004, Version 3: March 2005, Version 4: September 2005, Version 5 and Version 6: June 2006, Version 7: April 2007, Version 8 (including SVC extension): Consented in July 2007
43. ITU-T Rec. & ISO/IEC 14496–10 AVC.: Advanced video coding for generic audiovisual services, version 3 (2005)
44. Schwarz, et al. H.: Technical description of the HHI proposal for SVC CE1. ISO/IEC JTC1/WG11, Doc. m11244, Palma de Mallorca, Spain, October 2004
45. Reichel, J., Schwarz, H., Wien, M.: Scalable video coding joint draft 6 Joint video team, Doc. JVT-S201, Geneva, Switzerland, April 2006
46. Coding of audiovisual objectsPart 10: Advanced video coding, International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), ISO/IEC14 496–10 (identical to ITU-T Recommendation H.264)
47. Richardson, I.E.G.: Video Codec Design, John Wiley & Sons. Ltd. (2002). doi:10.1002/0470847832
48. Andreopoulos, Y., van der Schaar, M., Munteanu, A., Barbarien, J., Schelkens, P., Cornelis, J.: Complete-to-overcomplete discrete wavelet transforms for scalable video coding with MCTF. In: Proceedings SPIE/IEEE Visual Communication Image Process., pp. 719–731 (2003)
49. Andreopoulos, Y., Munteanu, A., Barbarien, J., van der Schaar, M., Cornelis, J., Schelkens, P.: In-band motion compensated temporal filtering. Signal Process. Image Commun. **19**, 653–673 (2004)
50. Wang, B., Loo, K.K., Yip, P.Y., Siyau, M.F.: A simplified scalable wavelet video codec with MCTF structure. In: International Conference on Digital Telecommunications, (ICDT'06), 29–31 August 2006. doi:10.1109/ICDT.2006.11
51. Andreopoulos, Y., van der Schaar, M., Munteanu, A., Barbarien, J., Schelkens, P., Cornelis, J.: Fully-scalable wavelet video coding using in-band motion-compensated temporal filtering. In: Proceedings IEEE International Conference Acoustical Speech and Signal Process., pp. III-417–III-420 (2003)
52. Park, H.-W., Kim, H.-S.: Motion estimation using low-band-shift method for wavelet-based moving-picture coding. IEEE Trans. On Image Process. **9**(4), 577–587 (2000)
53. Daubechies, I., Sweldens, W.: Factoring wavelet transforms into lifting steps. J. Fourier Anal. Apple **4**(3), 247–269 (1998)

Chapter 3
VLSI Architecture for Fast Three Step Search Algorithm

The goal of this chapter is to introduce a Fast Three Step Search (FTSS) algorithm, and its VLSI architecture. The chapter starts with a brief discussion on FTSS and three step search (TSS) algorithm. Section 3.2 presents the method by which one can predict the direction of current motion vector. The FTSS algorithm has been presented in Sect. 3.3. Section 3.4 gives the detailed VLSI architecture required for implementation of the FTSS algorithm. The simulation and the synthesis results of the proposed algorithm and the corresponding architecture are presented in the subsequent section. The conclusions are finally presented in the last section.

3.1 Introduction

In general, there is a correlation between the motion vector of current block and the motion vector of previous block [1]. This is particularly true in low bit-rate video applications, including videophone and video conferencing, where fast and complex movements are rarely involved. To reduce the heavy computational cost resulting from the massive number of candidate locations, three step search algorithm (TSS) searches for the best motion vector in a coarse-to-fine manner. In the first step, nine sparsely located candidates are evaluated and the one with the minimum SAD is picked out. In the second step, the search is focused on the area centered at the winner of the previous step, but distances between candidate locations are shortened by half. In the same manner, the third step compares SAD's of nine locations around the winner of the second step and then gives the final motion vector. For the commonly used search range of $d1 = d2 = 7$, the hierarchical search procedure decreases the number of search locations to 1/9 of the exhaustive approach. The Fast Three Step Search algorithm (FTSS) [2] makes use of the directional information from adjacent previous motion vector and unimodal error surface assumption (UESA). It determines the direction of the current motion vector from the previous motion vector and reduces the computation for checking the candidate motion vector using the UESA. The UESA means that the error increases monotonically in getting away from the global minimum [3].

I. Chakrabarti et al., *Motion Estimation for Video Coding*,
Studies in Computational Intelligence 590, DOI 10.1007/978-3-319-14376-7_3

3.2 Prediction of Direction of Current Motion Vector

Directional information of the previous motion vector was used for the estimation of the current motion vector. Direction of the current motion vector is predicted with the adjacent motion vector for the current frame. Figure 3.1 shows the determination of direction for the current motion vector from the sign of previous motion vector that is the motion vector of the left neighbor. Note that the positive direction of the vertical axis points downwards, while that of the horizontal axis points towards right. The set of search points (checking points) in the subsequent phase, when the previous motion vector is directed towards top left (as shown in Fig. 3.1a) is found out based on four exhaustive conditions as depicted in Fig. 3.2. The second phase of checking points for the remaining three possible previous motion vectors (as shown in Fig. 3.1b, c, d) will be determined in a similar manner. By selecting the direction from the adjacent previous motion vector instead of arbitrary search direction, the probability of occurrence of 4 checking points in the first step will be increased. That is, by exact estimation of direction of the current motion vector, we can check only four checking points in the first step, not five or six checking points as [3]. By using the directional information, we are able to decrease one or two checking points in the first step as compared with the Lu and Lious algorithm [3]. The number of checking points for the first step is nine in the original TSS algorithm. The FTSS algorithm reduces the number of checking points by 4 or 5 points for each step, while ensuring

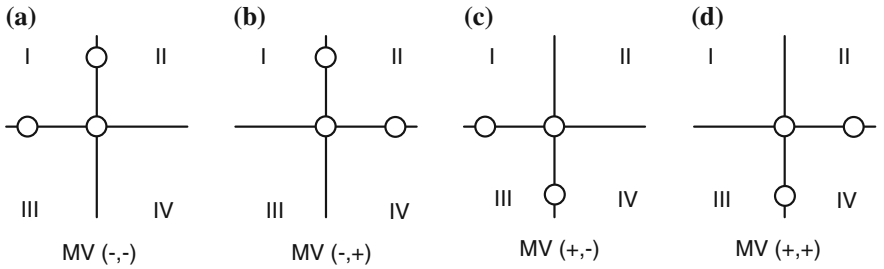


Fig. 3.1 Direction of the current MV from the previous MV

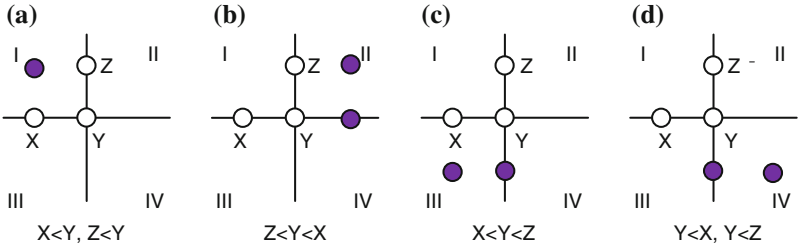


Fig. 3.2 Next checking points after checking 3 points

a performance close to TSS algorithm. The number of checking points for the first step is nine in the original TSS algorithm. The FTSS algorithm reduces the number of checking points by 4 or 5 points for each step, while ensuring a performance close to TSS algorithm.

3.3 Fast Three Step Search Algorithm (FTSS)

The FTSS algorithm [2] differs from the TSS algorithm [4] in the following ways:-

- Previous motion vector is considered in FTSS.
- For each step, in the first phase of FTSS algorithm, one needs to consider the three search points as specified in Fig. 3.1.
- In second phase, it considers one or two search point(s) as specified in Fig. 3.2.

So for each step it checks four or five search points. So the minimum number of search points for three steps is 12 and the maximum number of search points is 15. But in TSS Algorithm, the number of search points is fixed at 25. The arrangement of search points is shown in Fig. 3.3. Assume that the sign of the adjacent previous motion vector is (+, −). According to Fig. 3.1, first three search points are S4, S5 and S8. In the first step, we will compare SAD values of the S4, S5 and S8 (i.e. SAD (4), SAD (5), and SAD (8)). In this algorithm, SAD(x) is Sum of Absolute Difference at the location Sx. Search points (candidate locations) S1–S9 are shown in Fig. 3.3.

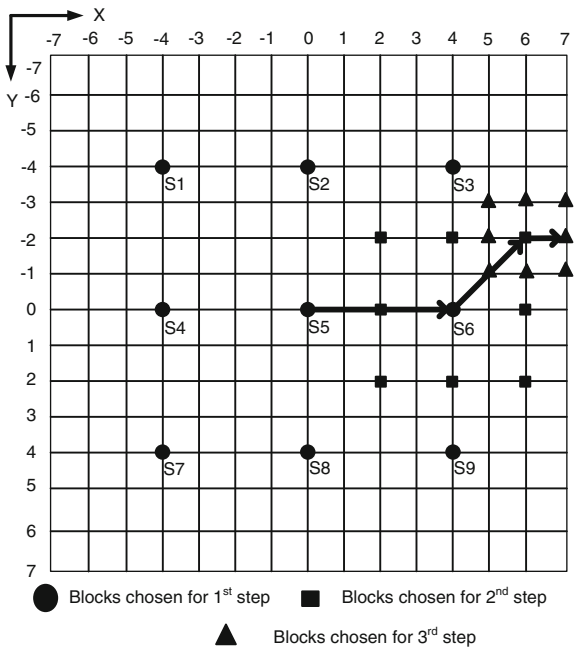


Fig. 3.3 The FTSS Algorithm (the 9 candidate locations in a step labeled by S1–S9)

Algorithm 3.1 Fast Three Step Search Algorithm

```
1: if SAD(6)< SAD(5) & SAD(8) < SAD(5) then
2: check 9 & select min [SAD(x)]                                  (4 Pts)
3: end if
4: if SAD(6) < SAD(5) < SAD(8) then
5: check 3 & select min [SAD(x)]                                  (4 Pts)
6: end if
7: if SAD(8) < SAD(5) < SAD(6) then
8: check 7 & select min [SAD(x)]                                  (4 Pts)
9: end if
10: if SAD(5) < SAD(6)& SAD(5) < SAD(8) then
11:    if SAD(5) < SAD(6) < SAD(8) then
12:       if SAD(2) < SAD(5) then
13: check 1 & select min [SAD(x)]                                 (5 Pts)
14:       else          min [SAD(x)] = SAD(5)                     (4 Pts)
15:       end if
16:    end if
17:    if SAD(5) < SAD(8) < SAD(6) then
18:       if SAD(4) < SAD(5) then
19: check 1 & select min [SAD(x)]                                 (5 Pts)
20:       else          min [SAD(x)]= SAD(5)                      (4 pts)
21:       end if
22:    end if
23: end if
```

3.4 Proposed 3-PE Architecture for FTSS

The proposed architecture for FTSS consists of memory sub system, control unit, and process control unit, as depicted in Fig. 3.4. Memory sub-system (MSS): It consists of three half search area buffers, which are used to store the search block from external memory [4]. Two half search area buffers are used for each task. And these buffers are again divided into nine memory modules to enhance memory bandwidth. This also enables memory interleaving for simultaneous accesses [5].

Control Unit (CU): It is a finite state machine. This is having a counter to count the number of clock cycles, and an algorithm unit to decide the second phase search points in every step. Depending on the number of clock cycles, it generates the timing and control signals for all the blocks. The major functions of the control unit are

- To activate the required half search area buffer to access the external memory.
- To send search point's base addresses to process control unit for each step.

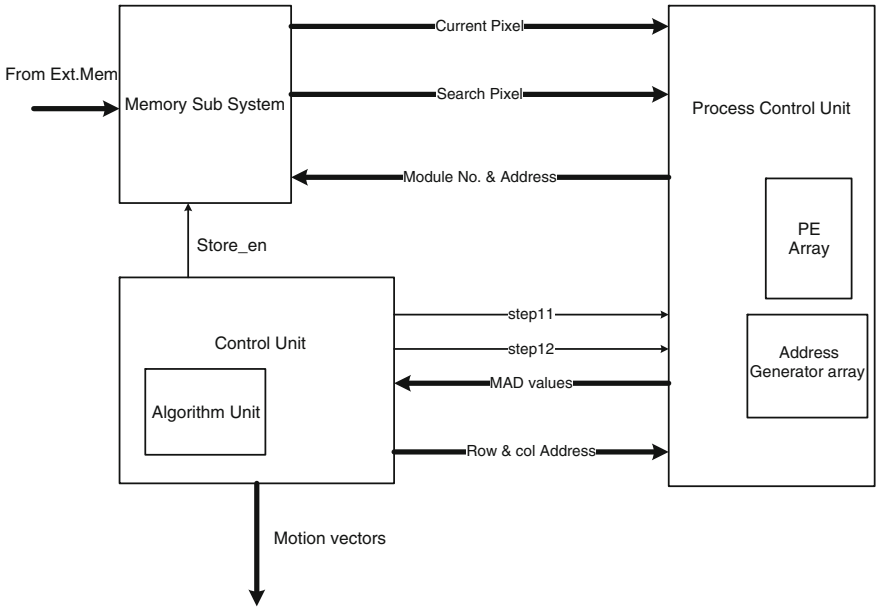


Fig. 3.4 Block diagram of proposed architecture for FTSS algorithm

- To take the SAD values from the process control block, according to the minimum SAD value select the next checking point(s) using algorithm block.
- To find the motion vector after completion of three steps.

Process control unit (PCU): It contains the processing element array unit (PE Array) and address generator array unit. The major tasks of the process control unit are listed as follows:

- To take step activation signal and base addresses of search points from the control unit. From base address and offset address, the address generator calculates memory module number and module address.
- To send module numbers and module addresses to memory sub system, then take search pixels from memory sub system, and current pixel from external (system) memory.
- To calculate SAD values for each search point and send to control unit.

Address generator array unit (AGU): It consists of three address generators. Each address generator generates module number and module address by using base address which is coming from control unit, and offset address which is generated internally. According to FTSS algorithm, each step is having two phases. Control unit will selects, 3 search points in 1st phase out of nine, according to sign of adjacent previous motion vector. Base addresses of the selected search points will send to the process control unit. After 256 clocks, process control unit will send the SAD values for those search points. According to SAD values of the 3 search points, algorithm

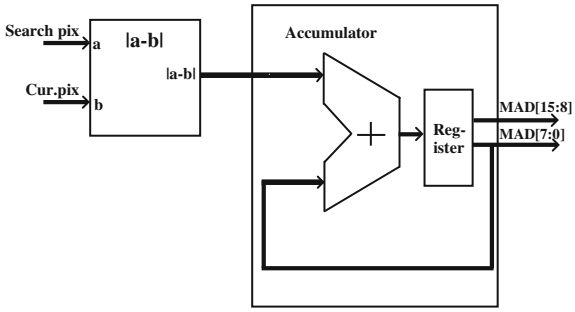


Fig. 3.5 Block diagram for processing element (PE)

unit decides the next one or two search points for 2nd phase. This will continue for remaining two steps. After three steps, the control unit generate the motion vector. The distance between search points for the first step is equal to 4, for second step, it is equal to 2 and for third step, it is 1.

PE array: It consists of three processing elements (PEs). Each PE takes a search pixel and a current pixel as input from memory sub system and finds mean absolute difference (MAD). The difference will be accumulated and produce the MAD value for each search point (16×16 blocks) for every 256 clocks. The structure of PE is shown in Fig. 3.5.

The proposed parallel architecture with 3 PE's is a good choice for the FTSS [2] algorithm. For, there are two phases in each step, according to FTSS algorithm specified in Algorithm 3.1. In the first phase in each step, the number of checking points are 3. In the second phase in each step the number of checking points are 1 or 2. So the number of parallel checking points never exceeds three, which entails only three PE's. However, the difficulties on data addressing and interconnection complexity make it hard to implement. The present chapter suggests a hardware structure that can effectively solve all these problems. This architecture is based on two data management techniques, namely (1) an on chip buffer configuration for reducing the number of external memory accesses, (2) the residual memory interleaving for parallel data accesses.

The basic 3-PE structure and its input data flow are shown in Fig. 3.6. For 16×16 blocks and a vector range of -7 to $+7$ pixels in both horizontal and vertical directions, this method in principle completes a block-matching every $256 \times 2 \times 3 = 1{,}536$ clock cycles. Furthermore, because the required pixels are sent to PE's in parallel without data skewing, the latency delay is also very short. In essence, this architecture provides a flexible high-speed motion estimator at a low cost. To reduce the loads of system memory, chip I/O, and interconnection, we used the on chip buffer configuration for data reuse and residual memory interleaving, as described in [4].

Random-access on-chip buffers have been used in the proposed architecture and utilized characteristics of FTSS to reduce the addressing and control overheads. The proposed architecture sequentially inputs current block pixels and broadcasts them to all PE's (as shown in Fig. 3.6). However, search area pixels are stored in on-chip buffers so that they are internally available whenever necessary. In general, a double

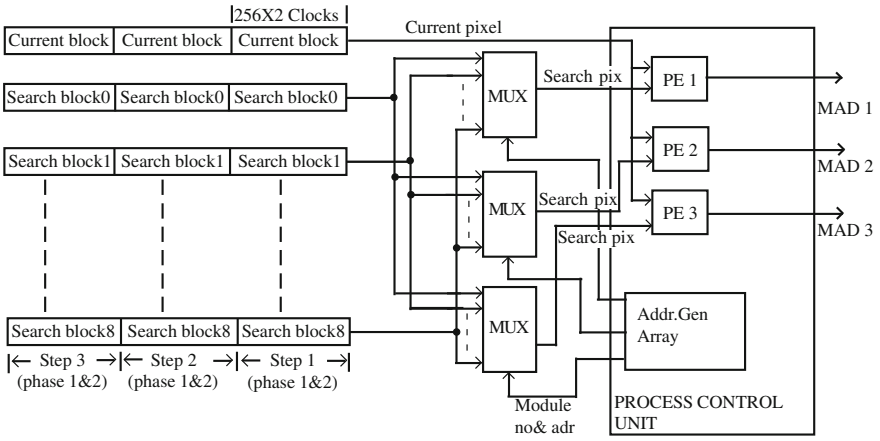


Fig. 3.6 Proposed basic 3-PE structure and its input data flow

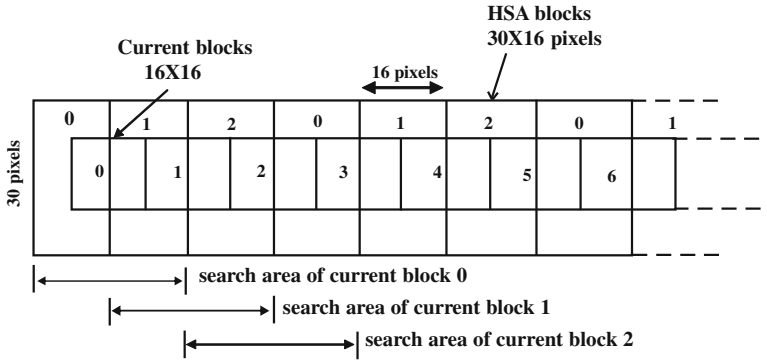


Fig. 3.7 Search areas of the adjacent current blocks

Fig. 3.8 Search area buffers used for each task

	T0	T1	T2	T3	T4	T5 → Task Ti
HSA Buffers	0	1	2	0	1	2
	1	2	0	1	2	0

sized buffer is required for simultaneous I/O and computation [6]. For further I/O bandwidth reduction, we utilize the overlap between search areas of adjacent current blocks as shown in Fig. 3.7, and utilized the scheme of three half-search-areas (HSA) as described in [4]. An HSA buffer stores an HSA block (30 × 16 pixels), which is larger than the real half search area (30 × 16 pixels). The extra column is for filling the gap between search areas of different tasks. The operations of the three HSA buffers are shown in Fig. 3.8. When executing task 0 (matching current block 0),

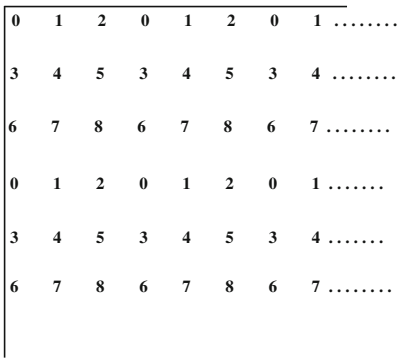


Fig. 3.9 Residual memory interleaving

PE's access search area pixels from HSA buffer 0 and 1. During these $256 \times 2 \times 3$ or 1,536 clock cycles, the HSA buffer 2 is being filled by the right-half of search area for task 1. In the next $256 \times 2 \times 3$ clock cycles, search area pixels are accessed from buffer 1 and 2 for executing task 1, and new data are input to buffer 0. Cyclic manner, the 30×16 new pixels can be easily accessed from system memory during $256 \times 2 \times 3$ or 1,536 clock cycles by using only one input port.

Memory interleaving has been used for simultaneous accessing of pixels which are required for the 3-PE architecture. Residual memory interleaving has been done by dividing the search area buffer into $3^2 = 9$ memory modules. The search area 1st row pixel 0 (top left) is loaded in to memory module M0, pixel 1 in to M1, pixel 2 is loaded into M2, pixel 3 is loaded into M0, and pixel 4 into M1 this repeats for entire 1st row. Second row pixels are stored in memory modules M3, M4 and M5. Third row pixels are stored in memory modules M6, M7 and M8. Again 4th row pixels are stored in memory modules M0, M1 and M2. This is repeated for entire search area. Memory module numbers corresponding to pixels are shown in Fig. 3.9.

3.5 Results

3.5.1 Simulation Results

The proposed architecture has been simulated in Xilinx-ISE 8.1i platform using vertex 4 device family. Synthesis tool was XST(VHDL/Verilog), and simulator was Modelsim-XE(verilog). Functionality of proposed architecture has been tested on two images, namely 'Lena' and 'News reader' of size 128×128. Block size has been taken to be 16×16,while the search range has been fixed at $[-7, +7]$. Figure 3.10 shows the number of checking points per block for TSS and FTSS. And Table 3.1 shows the performance comparison between 9 PE's TSS and 3 PE's FTSS.

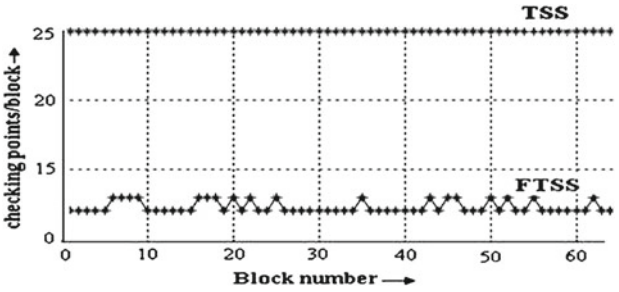


Fig. 3.10 Simulation result

Table 3.1 Performance comparison between FTSS and TSS in terms of PSNR

Image type	(3PEs) FTSS (dB)	(9PEs) TSS (dB)
trevor.qcif	37.1061	37.4403
News reader.qcif	34.592	34.812

3.5.2 Synthesis Results

Verilog language is employed to model the proposed architecture at behavioral domain. The behavioral verilog model is then used for logic circuit synthesis executed by invoking the Synopsys design vision synthesis tool. The logic circuit that implements a proposed architecture has been obtained. Table 3.2 shows the area requirements for proposed (3PEs) FTSS Architecture and (9 PEs) TSS architecture in terms of standard unit cells. Table 3.3 shows the power consumption for proposed (3PEs) FTSS Architecture and (9 PEs) TSS architecture. Negative percentage of saving in 2nd row (control unit) of Tables 3.2 and 3.3 indicates that the complexity of the control unit has been increased due to algorithmic unit in 3 PEs FTSS architecture. That is area and power requirements are more in control unit of 3 PEs FTSS architecture.

Table 3.2 Area in (μm^2) comparison between proposed (3PEs) FTSS and (9PEs) TSS architecture

Unit name	FTSS (3-PEs)	TSS (9-PEs)	% of saving
Control unit	16708.00	1568	−90.61
Process control	61446.00	166133.00	63.034
Total	78154.00	167701.00	53.39

Table 3.3 Power comparision between proposed (3PEs) FTSS and (9PEs) TSS architecture

Unit name	FTSS (3-PEs) (mW)	TSS (9-PEs)	% of saving
Control unit	2.43	475.2 uW	−80.5
Process control	50.6	83.01 mW	39.01
Total	53.0	83.49 mW	36.44

3.6 Conclusions

The present chapter has focussed on an efficient VLSI architecture for the FTSS algorithm. Configuration of random access on-chip buffer solves the problem of chip I/O and memory bandwidth requirements. The buffer and the input data have been arranged according to the principle of residual memory interleaving for parallel accessing of data. The proposed architecture has been designed with 3 PEs, and as compared with 9 PEs TSS architecture, the chip area has been reduced by almost 50 %. Moreover, the number of checking points has been reduced as compared with TSS and NTSS algorithms. This paves the way for reduced power consumption by almost 25 %. This architecture is considered to be useful for low bit rate, low power video applications like video telephony, video conferencing and HDTV.

References

1. Jain, J.R., Jain, A.K.: Displacement measurement and its application in interframe image coding. IEEE Trans. Commun. **COM–29**(12), 1799–1808 (1981)
2. Kim, J.-N., Choi, T.-S.: A fast three step search algorithm with minimum checking points. In: Proceedings of IEEE Conference on Consume Electronics, pp. 132–133, 2–4 June 1998
3. Lu, J., Liou, M.L.: A simple and efficient search algorithm for block-matching motion estimation. IEEE Trans. Circuits Syst. Video Technol. **7**(2), 429–433 (1997)
4. Jong, H.-M., Chen, L.-G., Chiueh, T.-D.: Parallel architectures for 3-step hierarchical search block-matching algorithm. IEEE Trans. Circuit Syst. Video Technol. **4**(4), 407–416 (1994)
5. Srinivasarao, B.K.N., Chakrabarti, I.: Parallel architectural implementation of the fast three step search algorithm for block motion estimation. In: Proceedings of 5th International Multi-Conference on Systems, Signals and Devices (SSD-08), pp. 1–6. Amman, Jordan, 20–22 July 2008. doi:10.1109/SSD.2008.4632849
6. Po, L.-M., Ma, W.-C.: A novel four step search algorithm for fast block motion estimation. IEEE Trans. Circuits Syst. Video Technol. **6**(3), 313–317 (1996)

Chapter 4
Parallel Architecture for Successive Elimination Block Matching Algorithm

The Successive Elimination Algorithm (SEA) effectively eliminates the search points within the search window and thus decreases the number of matching evaluation instances that require very intensive computations compared to the standard Full Search Algorithm (FSA). This chapter begins with an introduction to SEA followed by a detailed description of the SEA algorithm. Section 4.3 gives the details of parallel architecture for SEA algorithm. Relevant design statistics on area and power for comparing between SEA and FSA implementations are presented in the subsequent section. The conclusions are finally presented in the last section.

4.1 Introduction

Recently the market for portable multimedia applications, such as MPEG video camera, wireless video phone, and portable wireless multimedia terminal, has been on the rise [1]. Consequently, fast VLSI video compression processors are in high demand for the emerging wireless video applications. Typical video compression processors today include VLSI motion estimators which implement the FSA. In the block-matching motion estimation, the motion vector is the displacement of a macroblock with the minimum distortion from the reference macroblock. The FSA determines the motion vector by identifying a macroblock with the minimum distortion from a pool of all possible candidate blocks in the search area. The FSA thus offers the optimal solution; however, existing implementations of this algorithm are computationally expensive and time consuming because they typically compute the distortion values of all possible candidate macroblocks. Many motion estimation algorithms are found in the literature [2–4]. Some of them are fast but cannot guarantee an optimal solution; they can be stuck in local optima. Such algorithms are fast, and consume less power when implemented in VLSI; however, they can result in high levels of distortion that cannot be accepted in many applications [1, 5]. The disadvantage of these algorithms is that they result in sub-optimal solutions because the search space is reduced. These approaches reduce the computational load, and

I. Chakrabarti et al., *Motion Estimation for Video Coding*,
Studies in Computational Intelligence 590, DOI 10.1007/978-3-319-14376-7_4

consequently the power consumption, by sacrificing the optimality of the solution. Without sacrificing the optimality, the successive elimination algorithm (SEA) proposed by Li and Salari [6] reduces the computational load of the FSA. The motion vector found by SEA is identical to the motion vector found by FSA.

4.2 Successive Elimination Algorithm (SEA)

The aim of motion estimation is to find the best matching block of the reference block in the current Frame. The top left comer point of the best matching block exists within the search window in the previous frame. Sum of Absolute Difference (SAD), which is the error norm, is used to measure the matching between the two blocks. The best matching block has minimum SAD. SAD is defined as Eq. (4.1)

$$SAD(i, j) = ||X - Y(i, j)|| \tag{4.1}$$

In Eq. (4.1) X represents the reference block in the current frame for which a motion vector is required, and $Y(i, j)$ represents the possible candidate motion vector block within the search window. Note that (i, j) is the upper left comer point of the block, and (i, j) is the point within search window. In FSA, SAD is computed for every point (i, j) in the search window, one displacement at a time. As each SAD is calculated, SAD is compared against the current minimum SAD (cur_min_SAD). If it is smaller than the cur_min_SAD then it becomes the current minimum SAD. When this procedure is repeated for all of the points (i, j) in the search window, the block that has the current minimum SAD is the best matching block, and the displacement vector of the best matching block is the motion vector for the reference block in the current frame.

By using SEA, motion vector for each reference block in the current frame can be found with much less computational load than the exhaustive search algorithm by using mathematical property. Applying mathematical inequality $||A||_1 - ||B||_1 \leq ||A - B||_1$ for A = X and B = Y(i, j) gives

$$|||X||_1 - ||Y(i, j)|||_1 \leq ||X - Y(i, j)||_1 \tag{4.2}$$

where $||X|| = \sum_k |X_k|$ Note that X represents the reference block in the current frame for which motion vector is required, and Y(i,j) represents the possible candidate blocks within the search window. $||X||_1$,$||Y(i, j)||_1$ are sum norms and those are precomputed. Assume that, cur_min_SAD = SAD(m,n) = $||X - Y(m, n)||_1$ is calculated for an initial matching candidate block. To find the best matching block, the only interested is the blocks its SADs are less than cur_min_SAD. From the

Eqs. (4.1) and (4.2), the blocks its sum norm $||Y(i, j)||_1$ satisfy Eq. (4.3) can not become the best matching candidate block, therefore, $||X - Y(i, j)||_1$ calculation does not need. By using Eq. (4.3), many points in the search window can be eliminated by only calculating absolute difference between sum norms, $(|||X||_1 - ||Y(i, j)|||_1$ symbolized as SAD_SN) without involving calculation of $||X - Y(i, j)||_1$.

$$\text{cur\_min\_SAD} \leq |||\text{X}||_1 - ||Y(i, j)|||_1 = SAD\_SN \tag{4.3}$$

If Y(i, j) cannot satisfy Eq. (4.3), SAD (i, j) must be calculated. If SAD (i, j) is less than cur_min_SAD, SAD (i, j) becomes cur_min_SAD. When this procedure is repeated for all of the points (i, j) in the search window, the block that has the current minimum SAD(cur_min_SAD) is the best matching block, and the displacement vector of the best matching block is the motion vector for the reference block. If the upper left corner point of the best matching block is (u, v) and upper left corner point of the reference block in the current frame is (a, b), the motion vector is (u-a, v-b). The SEA speeds up the process of finding the motion vector by eliminating impossible candidate blocks in the search window before their SAD calculation that requires very intensive computations. SEA algorithm is expressed as follows.

Algorithm 4.1 Successive elimination algorithm

1. Select initial candidate motion vector block its upper left corner point is one of the search points within the search window in the previous frame.
2. Calculate SAD at the selected point.
3. cur_min_SAD = SAD
4. select another point among the rest of the search points within the search window
5. • calculate SAD_SN at the selected search point.
   • if (cur_min_SAD $\leq$ SAD_SN) go to 7
   • calculate SAD at the selected search point (the matching evaluation point)
6. if(SAD $<$ cur_min_SAD) then cur_min_SAD = SAD
7. if(all the points in the search window is not completed) then go to step 4
8. Minimum SAD = cur_min_SAD, find motion vector.

4.3 Proposed Parallel Architecture for SEA

Figure 4.1 shows the block diagram of the proposed architecture [7]. It consists of three major units, namely internal memory unit, control unit, and process control unit.

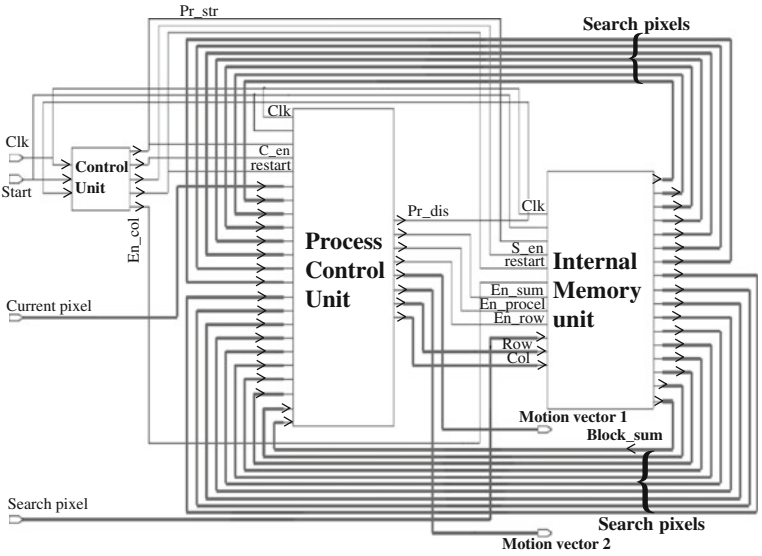


Fig. 4.1 Complete block diagram to a proposed parallel architecture

4.3.1 Internal Memory Unit (IMU)

The internal memory unit is consists of four search area buffers (SA buffers), and one register array called col_array. SA buffers are used to store the search block from external memory. Three SA buffers are used for each task. And these buffers are again divided into sixteen memory modules to enhance memory bandwidth. Each memory module in a buffer can store 48 pixels (i.e. one entire column in a search area). This also enables memory interleaving for simultaneous accesses. Col_array is array of 48 registers. Each register can store 16 bit data, which is used to store the sum of 16 pixels in a column.

4.3.2 Control Unit (CU)

The control unit is a finite state machine. It consists of a counter to count the number of clocks. Depending on the number of clocks, it generates the timing and control signals for all the blocks. When inputs 'Start' and 'Pr_dis' goes to logic 1 then counter is initialized to zero, otherwise it continuously counts the number of clocks.

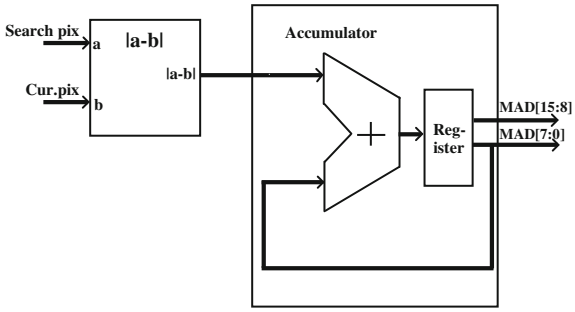


Fig. 4.2 Block diagram for processing element (PE)

4.3.3 Process Control Unit (PCU)

The process control unit consists of processing element array (PE array), and Address generator unit. It calculates the SAD value of the search point if it is necessary. And it calculates the motion vector from the search point having minimum SAD value. The PE array consists of 16 processing elements (PEs). Each PE takes a search pixel and a current pixel as inputs and found the partial Sum of absolute difference (PSAD). After 16 clocks sum of all 16 PSADs will give the SAD. The structure of PE is shown in Fig. 4.2.

4.3.4 Working of the Proposed Architecture

Given a block of size $N \times N$, the block motion estimation searches for a search point that yields the minimum SAD value within a neighborhood. Suppose that the maximum motion in vertical and horizontal direction is 2N. We consider the search block of size 48×48 and reference block (current block) of size 16×16.

The present chapter suggests a hardware structure based on two data management techniques, namely (1) an on chip buffer configuration for reducing the number of external memory accesses, (2) the residual memory interleaving for parallel data accesses [8]. Furthermore, because the required pixels are sent to PEs in parallel without data skewing, the latency delay is also very short. In essence, this architecture provides a flexible high-speed motion estimator at a low cost. To reduce the loads of system memory, chip I/O, and inter-connection, we used the on chip buffer configuration for data reuse and residual memory interleaving, as described in [2].

The proposed architecture sequentially inputs current block pixels and search pixels. For a task 0 (i.e. finding the Motion vector for current block 0) Search pixels are stored in SA buffer 0, 1, and 2. Figure 4.3 shows the SA buffers corresponding to each task. Figure 4.4 shows the memory modules of each SA buffers, corresponding to each pixel. Pixel (0, 0) is stored in module 0 (M0) of buffer 0, pixel (0, 1) is stored

	T0	T1	T2	T3	T4	T5	⟶Task Ti
Search area Buffers	0	1	2	3	0	1	
	1	2	3	0	1	2	
	2	3	0	1	2	3	

Fig. 4.3 Search area buffers for each task

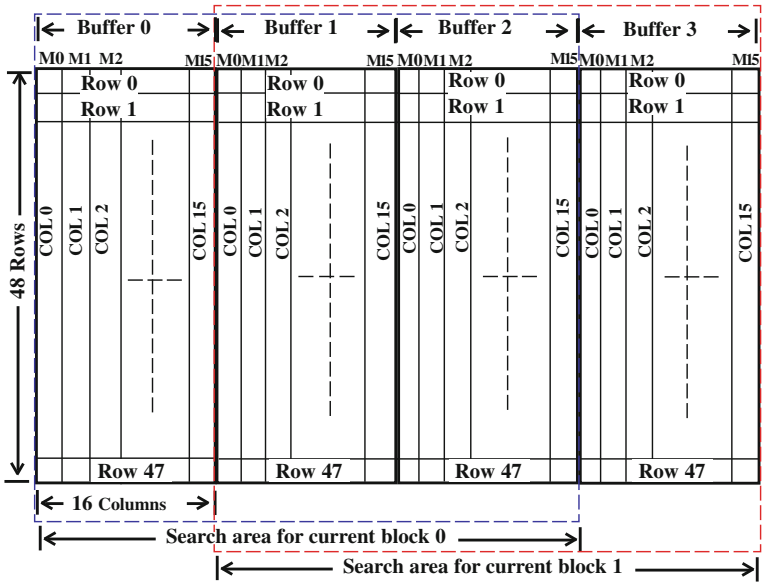


Fig. 4.4 Search area buffers and its memory modules

in module 1(M1) of buffer 0, and pixel (0, 47) is stored in module 15 (M15) of buffer 2. Figure 4.4 shows that same column pixels are stored in same memory module. None of the same row pixels are available in one module. So while calculating the SAD value of a particular search point all the same row pixels we can access in parallel for 16 PEs. Because required 16 columns pixels of same row are available in 16 different modules. Like wise current block (16 × 16) pixels are also stored in current_buffer of capacity 16 memory modules each module can store 16 pixels.

Assume for task 0, search area pixels are available in SA buffers 0, 1, and 2, and current block pixels are available in current_buffer which is available in process control unit. For first clock pulse, registers in a col_array loaded with 0th row pixels of search area. Each column pixel is loaded in to corresponding register, i.e. *i*th column pixel is loaded in to register (i). For the next clock pulse all the columns of next

row pixels are added with the previous pixels which are available in corresponding registers. Like that after 16 clocks each register is having sum of 16 pixels belongs to same column (i.e. register (i) is having sum of first 16 pixels of *i*th column). For the next 16 clocks sum of first 16 registers in col_array is available in block_sum register.

For FSA we need to calculate SAD value for all the search points ($2N \times 2N = 32 \times 32$ points). But according to SEA we need not to calculate SAD for all the search points. We can skip the calculation of SAD value by using Eq. (4.3). For left most top pixel (pixel (0,0)) we need to calculate the SAD value. Proposed architecture takes 16 clocks to calculate the exact SAD value. For calculating the SAD value, process control unit (PCU) send address of the pixel (i, j) (i.e. co-ordinate of the pixel) to internal memory unit (IMU). Then IMU sends search pixels from (i, j) to (i, j + 15), after that PCU sends these 16 search pixels as one input for each processing element (PE) in the PE array. And 16 current block pixels are coming from the current_buffer which is available internally as a second input for each PE. After 16 clocks sum of 16 PEs will give the SAD value for the search point (i, j), this will consider as a Cur_min_SAD value. reference_sum is sum of all pixel values in the current_buffer. The procedure followed by the proposed architecture for processing the remaining search points is given below.

1. temp = Block_sum - register(i) + register(i+16)
2. SAD_SN = abs(temp - reference_sum);
3. **If** (cur_min_SAD < SAD_SN)
   Then Go to next search point
   Else Go to calculation of SAD value
4. **If** (SAD < cur_min_SAD)
   Then cur_min_SAD = SAD;
   Else go to next search point
5. **If** (Row value of search point changed from (i-1) row to i^{th}row)
   Then {
   register(m) = register(m)-pixel(i-1, m) + pixel(i+15, m);
   Block_sum = register(m) + register(m+1) + ...register(m+15);
   }
   ElseIf (all the points in the search window is not completed)
   Then go to next search point
   Else Minimum SAD = cur_min_SAD, find motion vector.

For calculating the SAD_SN it requires only one clock. But to calculate exact SAD it requires 16 clocks. This process is continued till all the search points are completed. After completion of all the search points PCU finds the motion vector from cur_min_SAD. However, search area pixels are stored in on-chip buffers so that they are internally available whenever necessary. For further I/O bandwidth reduction, we utilize the overlap between search areas of adjacent current blocks (as shown in Fig. 4.5) and utilized the scheme of three search area buffers (SA buffer). It was

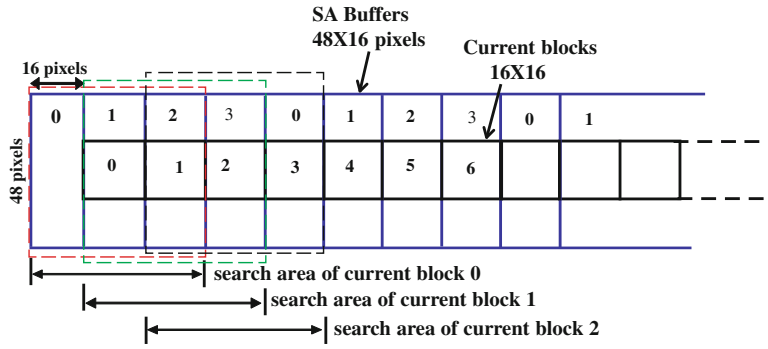


Fig. 4.5 Search area buffers for each current block

described in [9]. For executing task 1, search area pixels are accessed from buffer 1, 2 and 3, new data are input to buffer 0. In this cyclic manner, new pixels can be easily accessed from system memory during the task 0, by using only one input port.

4.4 Results

4.4.1 Simulation Results

The proposed architecture has been simulated in Xilinx-ISE 8.1i platform. Functionality of the proposed architecture has been tested on two benchmark video clippings namely, 'foreman.qcif' and 'Newsreader.qcif'.We have tested the five frames in each clipping; each frame is of size 176 × 144. Each frame contains 11 × 9 = 99 reference blocks of size 16 × 16. For the FSA no. of search points for each reference block is 32 × 32 = 1,024. No. of clocks required for the proposed architecture to find a motion vector to FSA is equals to no. of search points multiplied by 16 (i.e. 1,024 ∗ 16 = 16,384 because 16 clocks are required to calculate the SAD value). No. of clocks required to SEA is equals to no. of search points for which complete SAD value is calculated ∗ 16 + no. of points checked only the block_sum + no. of rows ∗ 16. Table 4.1 shows the no. of clocks and performance comparison between FSA and SEA for the inputs newsreader.qcif and foreman.qcif. Simulation

Table 4.1 Number of clocks and performance comparison between FSA and SEA for the input News reader sequence (5 frames, QCIF:176 X144 pixels, 8 bits per pixel)

	FSA	SEA	Percentage of speed increased
No. of SADs calculated per Frame	101,376	36,908	
PSNR	39.08 dB	39.08 dB	
Average no. of clocks required for one block	16,384 (1,024 × 16)	7,100	56.66 %

results shows that the average speed to calculate the motion vector by the proposed architecture for SEA is increased by nearly 59 % when compared to FSA.

4.4.2 Synthesis Results

Verilog language [10, 11] is employed to model the proposed architecture at behavioral domain. The behavioral Verilog model is then used for logic circuit synthesis executed by invoking the Synopsys design vision synthesis tool. The logic circuit that implements a proposed architecture has been obtained. Table 4.2 shows the area requirements for proposed Architecture for SEA and FSA, in terms of standard area units (area required by the 2-input NAND gate equals to 4 area units). The area requirement shows that Architecture for SEA will take nearly 30 % more than the FSA architecture. Table 4.3 shows the power consumption for proposed Architecture for SEA and FSA. That shows power requirements are nearly same for both architectures. The proposed architecture for SEA increased the speed by nearly 59 % with same power requirement and slight raise in area required by the FSA architecture. The proposed architecture can work up to 100 MHz frequency (Table 4.4).

Table 4.2 Area (in μm^2) comparison between architectures of SEA and FSA

Unit name	SEA	FSA
Memory	3213950.00	2216900.00
Control unit	848.00	793.00
Process control unit	87476.00	85073.00
Total	3302274.00	2302766.00

Table 4.3 Number of clocks and performance comparison between FSA and SEA for the input Foreman sequence (QCIF:176 $\times$144 pixels, 8 bits per pixel, 5 frames)

	FSA	SEA	Percentage of speed increased
No. of SADs calculated per Frame	101,376	30,959	
PSNR	38.46 dB	38.46 dB	
Average no. of clocks required for one block	16,384 (1,024$\times$16)	6,200	62.15 %

Table 4.4 Power comparison between architectures of SEA and FSA

Unit name	SEA	FSA
Memory	320.6612 mW	318.9846 mW
Control unit	101.5602 μW	96.0931 μW
Process control unit	17.4026 mW	17.2832 mW
Total	338.165 mW	337.2287 mW

4.5 Conclusions

The present chapter has focused on developing a parallel architecture for SEA. The proposed architecture for SEA increased the speed by nearly 59 % with same power requirement and accuracy, but slight increase in area required by the FSA architecture. Simulation study shows that the average search points are reduced by SEA to 70 % of the search points required by the FSA. Configuration of random access on-chip buffer solves the problem of chip I/O and memory bandwidth requirements. The buffer and the input data have been arranged according to the principle of residual memory interleaving for parallel accessing of data. This architecture is considered to be useful for real-time video applications like video telephony, video conferencing and HDTV.

References

1. Do, V.L., Yun, K.Y.: A low-power VLSI architecture for full-search block-matching motion estimation. IEEE Trans. Circuits Syst. Video Technol. **8**(4), 393–398 (1998)
2. Jong, H.-M., Chen, L.-G., Chiueh, T.-D.: Parallel architectures for 3-step hierarchical search block-matching algorithm. IEEE Trans. Circuits Syst. Video Technol. **4**(4), 407–416 (1994)
3. Srinivasan, R., Rao, K.: Predictive coding based on efficient motion estimation. In: Proceedings IEEE ICC'84, pp. 521–526 (1984)
4. Pun, A., Hang, H.M., Schilling, D.L.: An efficient block matching algorithm for motion compensated coding. In: Proceedings International Conference Acoustics, Speech, and Signal Processing, pp. 25.4.1–25.4.4 (1987)
5. Yeo, H., Hu, Y.: A novel matching criterion and low power architecture for real-time based block based motion estimation. In: Proceedings ASAP'96, pp. 122–130, August 1996
6. Li, W., Salari, E.: Successive elimination algorithm for motion estimation. IEEE Trans. Image Process. **4**(1), 15 Jan 1995
7. Srinivasarao, B.K.N., Chakrabarti, I.: A parallel architecture for successive elimination block matching algorithm. In: Proceedings TENCON-2008 (IEEE Region 10 Conference), pp. 1–6. Hyderabad, India, 19–21 November 2008
8. Mahmoud, H.A., Bayoumi, M.A., Wilson, B.: A low power architecture for a new efficient block matching motion estimation algorithm. In: Proceedings 43rd IEEE Midwest Symposium on Circuits and Systems, Lansing, 8–11 August 2000
9. Baglietto, P., Maresca, M., Migliaro, A., Migliardi, M.: Parallel implementation of the full search block matching algorithm for motion estimation. In: ASAP'95, pp. 182–192. IEEE Computer Society Press, July 1995
10. Palnitkar, S.: Verilog HDL: A Guide to Digital Design and Synthesis, 2nd edn. Prentice Hall, February 2003
11. Bhasker, J.: Verilog HDL Synthesis: A Practical Primer. Star Galaxy, November 1998

Chapter 5
Fast One-Bit Transformation Architectures

This chapter begins with an introduction to one-bit transformation process followed by its combination with the Diamond Search (DS) algorithm leading to a fast binary ME procedure. The novel data flow analysis for the DS algorithm has been provided in the subsequent section. Next, the proposed ME architectures based on the combination of the DS algorithm with one bit transformation have been presented. Relevant results dealing with synthesis of the proposed architectures are given next. Conclusions are finally drawn in the last section.

5.1 Introduction

Due to the stringent requirements of the real time video playback systems, video coding is the most essential part of any visual application. Furthermore, due to the limited channel and storage capacity, these applications require a very high compression ratio as well. Motion estimation (ME), which is the most essential part of any video coding technique, exploits and tries to minimize the temporal redundancy present between successive frames. ME, which is computationally intensive, involves about 80 % of the total computational power of the encoder [1]. Block matching motion estimation (BMME) is one of the most efficient and popular techniques to remove the temporal redundancy present between successive frames [2]. In this method, given two blocks of pixels, a source block of size b $\times$ b known as macro-blocks (MBs) and a search window larger than the source block, find the b $\times$ b sub-block in the search window that is closest to the source block in a previously available frame known as the reference frame. Motion vector (MV) is defined as the displacement between the current block position and the best matched one in the reference frame. This process is repeated until MV is found for all the blocks in the current frame. In order to improve the coding efficiency, variable blocks size (VBS) ME has been adopted in modern coding standards like H.264.

I. Chakrabarti et al., *Motion Estimation for Video Coding*,
Studies in Computational Intelligence 590, DOI 10.1007/978-3-319-14376-7_5

Let p denote the source block of b × b pixels, with $p_{i,j}$ being the pixel at row i and column j. Similarly, let w denote the search window with $w_{i,j}$ being the pixel at row i and column j. The sub-block of w at position x, y is denoted by $w^{x,y}$ and is the block of b × b pixels $w_{x+i,y+j}$, for $i = 1, 2, \ldots, b$, $j = 1, 2, \ldots, b$. The matching criterion of the BMME method has a direct impact on the coding efficiency and computational complexity. The distance between two blocks u and v can be measured in many metrics, e.g., mean squared error, sum of absolute differences (SAD), pel difference classification etc. [3]. But typically the mean absolute deviation is used. The mean absolute deviation or l_1 metric is given by

$$\|u, v\|_1 = \frac{1}{b^2} \sum_{i,j} |u_{i,j} - v_{i,j}| \tag{5.1}$$

Whatever may be the matching criterion, evaluation of the matching criterion on pixels with 8 bits/pixel representation requires a huge amount of computation. The computational load can be reduced to a great extent by representing the pixels with a reduced number of bits. As proposed in [4], an image frame with 8 bits/pixel representation is first converted into a binary frame with 1 bit/pixel representation. ME is then carried out on these binary image frames. Boolean exclusive OR (XOR) operation is used to find the number of non-matching points (NNMP). In this method, NNMP is regarded as the matching criterion in place of the conventional SAD.

In literature, many one-bit transformation (1-BT) kernels are available that convert an image with 8 bits/pixel representation into a binary image with 1 bit/pixel representation. The first 1-BT kernel was proposed by Natarajan et al. [4]. A multiplication free one-bit transformation (MF-1BT) kernel has been proposed in [5]. Binary ME with an early termination scheme have been proposed in [6]. Two bit transformation (2-BT) has been proposed in [7] for an enhanced accuracy in ME. The constrained one bit transformation (C-1BT) has been proposed in [8] for a better performance than 1-BT, but at the same time with a reduced complexity than 2-BT. The implementation of 1-BT based ME on hardware has been proposed for the first time by Natarajan et al. [4]. A high performance VBS ME architecture for MF-1BT has been proposed in [9]. In [10], MF-1BT and C-1BT ME architectures for fixed block size (FBS) are presented. Both the architectures [4, 10] are based on 16 processing elements (PEs).

The application of fast search algorithms like diamond search (DS) on 1-BT frames can further reduce the computational load to a great extent as compared to applying full search (FS) on 1-BT frames. The combination of 1-BT with DS [11] results in a very small degradation in the peak signal to noise ratio (PSNR) as compared to FS.

In this chapter, we have presented an in depth analysis of performing ME on 1-BT binary frames by applying DS algorithm and proposed architectures for 1-BT based FBS and VBS motion estimation. In the architectures presented in this chapter, the pixels from the current block (CB) are read only once from the external memory and are stored into the local memories of the ME hardware. Thereafter, the pixels from the CB are supplied to the appropriate PEs periodically. The overlap between the neighboring search locations for DS is also exploited in a novel way to reduce the number of external memory accesses. The architectures explained in this chapter are faster than a recently reported 1-BT based ME architecture [9].

5.2 One Bit Transformation and Diamond Search Algorithm

One bit transformation based ME algorithms reduce the computational complexity of the ME process to a great extent. The diamond search algorithm requires much less computational power when compared with FS based ME. At the same time, diamond search algorithm provides acceptable image quality, and therefore, is one of the most preferred search algorithms. In the present section, explains the functionality of 1-BT based ME and diamond search algorithm.

5.2.1 One Bit Transformation Based ME

Before applying 1-BT based ME, the original image frames having 8 bits/pixel representation is converted into binary image frames with 1 bit/pixel representation. This is done by filtering the original image frame by a multi band-pass filter. The filtered image is then compared with the original image to obtain the binary image. In the original work by Natarajan et al. [4], the kernel used for filtering was having 25 non-zero elements and required expensive floating point multiplications. MF-1BT has been proposed in [5] in which the complex floating point multiplications were replaced by simple shifting operations. This new kernel 'K' in matrix form has been shown in (5.2).

As in this kernel the normalization factor is a power of 2, the filtering can now be performed by mere shifting without any expensive multiplication operation. The original frame F is filtered by convolving it with K and the filtered frame $\hat{F}$ is obtained and then one-bit image frames are constructed.

$$K = \frac{1}{16}\begin{bmatrix} 0000000001000000000 \\ 0000000000000000000 \\ 0000000000000000000 \\ 0000001000001000000 \\ 0000000000000000000 \\ 0000000000000000000 \\ 0001000001000001000 \\ 0000000000000000000 \\ 0000000000000000000 \\ 1000001000001000001 \\ 0000000000000000000 \\ 0000000000000000000 \\ 0001000001000001000 \\ 0000000000000000000 \\ 0000000000000000000 \\ 0000001000001000000 \\ 0000000000000000000 \\ 0000000000000000000 \\ 0000000001000000000 \end{bmatrix} \tag{5.2}$$

$$B(i,\, j) = \begin{cases} 1, \text{ if } \mathrm{F(i,j)} > \hat{\mathrm{F}}\mathrm{(i,j)} \\ 0, \text{ otherwise} \end{cases} \tag{5.3}$$

Here, i and j are the spatial coordinates of the pixel. The foregoing process by which an original frame with 8 bits/pixel representation is converted into a binary frame with 1 bit/pixel representation is known as one-bit transformation. After this operation, the number of non-matching points (NNMP) at any point (m, n) for a MB of size N × N is found as:

$$NNMP(m,n) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} B^{t}(i,\, j) \oplus B^{t-1}(i+m,\, j+n) \tag{5.4}$$

where, $-s \le m, n \le s$.

Here, 's' is the maximum search range and $\oplus$ denotes XOR operation. Also, B^t and B^{t-1} represent the current and the reference 1-BT frames respectively.

5.2.2 Diamond Search Based 1-BT ME

In DS algorithm [12], the search pattern forms a diamond like shape as shown in Fig. 5.1. Also, there is no limit on the number of steps that may be involved in the algorithm. DS uses two fixed search patterns, where one is the large diamond search pattern (LDSP) and the other is the small diamond search pattern (SDSP). The search starts with the LDSP with its center of search located at the origin. The matching

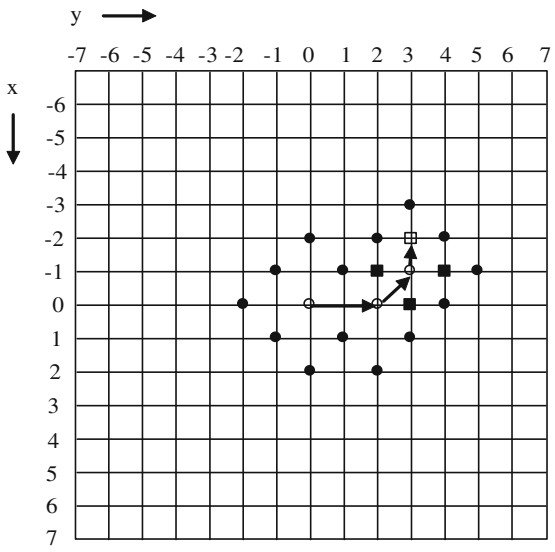


Fig. 5.1 An example illustrating the search strategy of DS algorithm

criterion is evaluated for all the nine points of LDSP and if the best match is found at the center, then the search pattern is switched to SDSP; else, another LDSP is selected with the new search center pointing to the location where the best match is found. This process is repeated until the best match is found at the center of the LDSP, after which the search pattern is switched to SDSP where the matching criterion is evaluated for four points around the center. The final MV is then found from the point giving the best match at this step.

Figure 5.1 demonstrates pictorially the steps required to find the motion vector $(-2, +3)$. The solid spheres indicate the location of the search points for LDSP. The solid squares indicate the location of the search points for SDSP. The locations of the minimum SAD points are indicated by transparent sphere (square) for LDSP (SDSP).

As DS algorithm is highly center biased, it is very fast as compared to FS algorithm. On the other hand, unlike other fast search techniques such as the new three step search, where the number of search steps are fixed, in DS the number of search steps are not fixed and thus its performance is very much close to that of FS in terms of the PSNR [13].

Motion estimation is usually performed on 1-BT frames by full search only [9, 10]. In the present work, several fast search techniques (e.g. three step search TSS), new three step search (NTSS), four step search (4-SS), and diamond search (DS)) have been applied on 1-BT frames. The performance of fast ME on 1-BT frames is evaluated based on PSNR and the average number of search points.

In Table 5.1, the PSNR values obtained by applying different fast search algorithms on 1-BT frames have been shown for different benchmark video sequences. All the sequences are in CIF (352×288) format, and each of the sequences contains 300

Table 5.1 Performance comparison in terms of PSNR (dB)

Video sequence		Full search	TSS	NTSS	4-SS	DS
Foreman	Max.	32.208	29.678	31.816	31.078	31.960
	Min.	28.106	25.900	27.652	27.965	27.899
	Avg.	30.766	28.982	29.651	30.031	30.458
Hall Monitor	Max.	34.388	31.631	33.599	34.163	34.478
	Min.	30.423	28.161	29.89	30.318	30.33
	Avg.	33.755	31.129	32.535	33.16	33.675
Football	Max.	23.262	22.435	22.16	22.616	22.966
	Min.	17.004	16.189	16.891	16.868	16.969
	Avg.	21.639	19.816	20.953	21.101	21.448
Tennis	Max.	30.465	28.899	29.997	30.108	30.169
	Min.	22.307	21.115	21.169	21.076	21.987
	Avg.	28.387	26.158	27.586	27.618	28.178
Coastguard	Max.	31.672	29.178	30.661	31.306	31.217
	Min.	23.191	21.806	22.898	22.165	22.766
	Avg.	29.512	27.638	28.76	28.898	28.922

Table 5.2 Average number of search points per MV generation

Video sequence	NTSS	4-SS	DS
Foreman	30.15	26.80	24.00
Hall Monitor	22.68	20.43	13.30
Football	22.56	21.58	17.70
Tennis	20.59	21.18	19.96
Coastguard	17.50	16.53	13.85

The number of search points for TSS and FS are fixed, namely 25 and 1,089 respectively for a search range of $[-16, 16]$

frames. The search range is taken as $[-16, 16]$ along both the axes. The average number of search points required to generate a MV can be regarded as a metric to measure the computational complexity for block matching. The average number of search points for different search algorithms on different video sequences have been shown in Table 5.2.

It can be observed from Table 5.1 that the combination of DS and 1-BT displays PSNR performance similar to the application of FS on 1-BT in most sequences with less than 0.21 dB degradation except for the sequences with complex motion like Foreman and the Coastguard for which the performance degrades by 0.31 and 0.59 dB respectively. On the other hand, it can be seen from Table 5.2, that DS based 1-BT ME always provides faster results than other fast search techniques. Taking all these observations into account, it can be inferred that applying DS on 1-BT frames provides almost the same performance as that of the FS based 1-BT ME, but at much lower computational complexity.

5.3 Data Flow Analysis for DS Algorithm

Due to the regularity in data-flow, Full Search Motion Estimation (FSME) is generally preferred from the implementation point of view. However, due to its high computational power requirements, many fast but sub-optimal search algorithms have been proposed. These fast search algorithms can reduce the computational power by up to 60 %, but as a side effect introduce irregular data flow [14–16]. In our experiment, it has been found that DS based 1-BT ME provides acceptable quality at much lower complexity than Full Search (FS) based 1-BT ME. Therefore, DS algorithm is one of the most preferred fast search algorithms. On the other hand, as DS introduces irregular data flow, the implementation of DS in hardware imposes considerable challenge for a VLSI designer. The search points for DS algorithm are arranged in a diamond like shape. This makes memory access mechanism for DS to be completely different from the FSME. The technique by which the data reuse is done for FS cannot be applied to DS in the same way. It has been shown in [17, 18] that the memory access is the most costly operation in ME hardware. Therefore, it is possible to reduce the overall power consumption of the ME hardware by reducing the number of external memory accesses. In the present section, a novel data-flow has been presented, which reduces the power consumption by maximizing the data reuse. In Fig. 5.2, the center of search is represented by location '4', and all the eight search locations around the center are represented by indexes starting from '0' to '8'. In order to understand the data overlap among different search locations, let us consider the size of the block as 4×4, and the center of search is located at pixel coordinates (2, 2). Now,

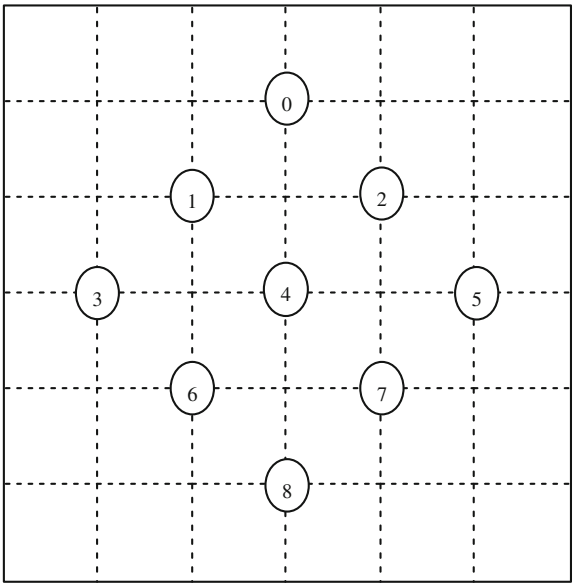


Fig. 5.2 Search locations for DS algorithm. The center of search is denoted by '*4*'

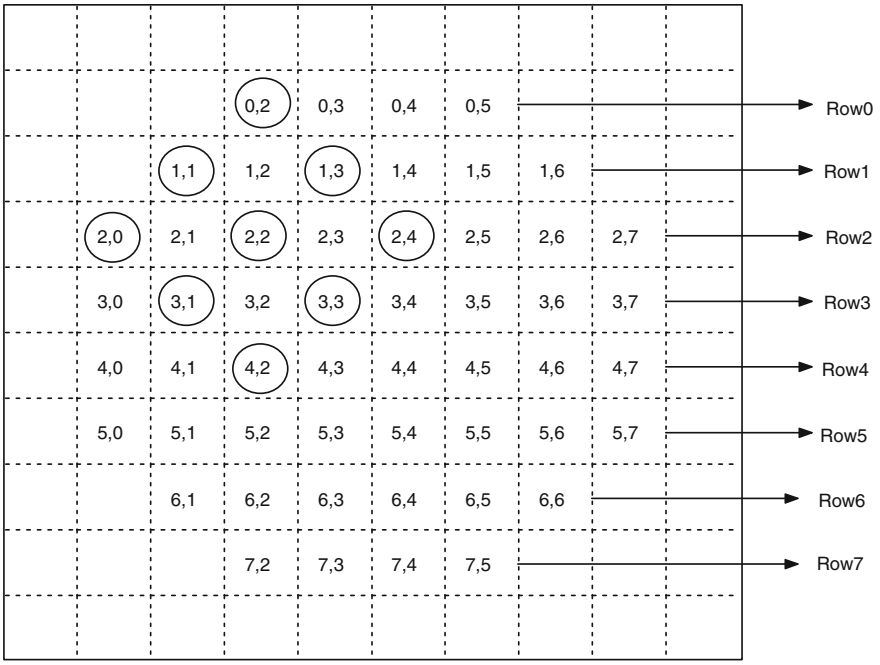


Fig. 5.3 Data flow example for DS with pixel coordinate (*2, 2*) as the center of search

with reference to Figs. 5.2 and 5.3, location '0' corresponds to the pixel coordinates (0, 2), location '1' corresponds to the pixel coordinates (1, 1), and so on.

In order to reduce the processing time for 1-BT based ME, it is desirable to process an entire row of data from the current block as well as the search block simultaneously [4, 10]. In that case, one has to access an entire row of data from both of the blocks simultaneously. Now, with reference to Figs. 5.2 and 5.3, one can observe that the search pixels for the entire first row corresponding to the location '0' in Fig. 5.2 have the pixel coordinates (0, 2), (0, 3), (0, 4) and (0, 5) in Fig. 5.3. It is also obvious from Fig. 5.3 that the first row of search pixels corresponding to the location '0' can be obtained from the same row, namely 'Row0'. Similarly, the search pixels for the entire first row corresponding to the search locations '1' and '2' can be obtained from the same row, namely Row1. In a similar way, 'Row2' provides the search pixels for the entire first row corresponding to the search locations '3', '4', and '5'. Similarly, the first row of search pixels for the search locations '6' and '7' are obtained from Row3 and lastly, the first row search pixels for the search location '8' are obtained from Row4. Thus, by exploiting the overlap of the search pixels for different search locations, one can reduce the number of memory accesses to 5 (Row0–Row4) instead of 9 as would have been required for a straightforward implementation.

It may be noted that the second row corresponding to the search location '0' in Fig. 5.2 has the pixel coordinates (1, 2), (1, 3), (1, 4), and (1, 5) in Fig. 5.3. Thus, the second row of data corresponding to the search location '1' can be obtained from

'Row1' (as shown in Fig. 5.3) which has already been read in the last step. Similar is the case for all the search locations except for the location '8', for which an entire new row (i.e. Row5) has to be read. Thus, only a new row of data has to be read in the subsequent steps of ME. Therefore, by exploiting the overlaps of the search pixels for different search locations the number of memory accesses for search pixels can he reduced down to 8 $(5 + 1 + 1 + 1)$, whereas 36 (9×4) memory accesses (corresponding to the fact that there exist 9 search locations and the block size is 4×4 as mentioned earlier in the present section) would have been required for a straightforward implementation of DS algorithm.

5.4 Proposed VLSI Architecture for 1-BT Based Fixed Block Size Motion Estimation

The block diagram of the proposed architecture for performing Fixed Block Size (FBS) ME on binary frames by applying DS has been shown in Fig. 5.4. The architecture includes 8 RAMs for storing all the pixels from the search window (SW), a register array for storing the current pixels, a register array for storing the search pixels termed as the search register array, a data selector array, nine PEs, a comparator and a process control unit. In the proposed scheme, the evaluations of the matching criterion for all the 9 search locations shown in Fig. 5.2 are done in parallel. As there are 9 search locations for DS algorithm, a total of 9 PEs are used.

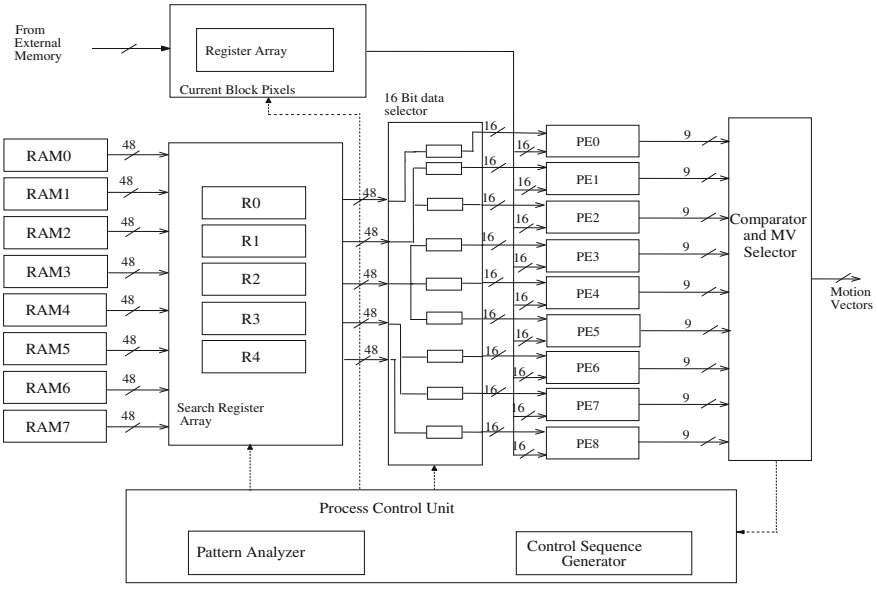


Fig. 5.4 Overall VLSI architecture for the proposed FBS ME

Before the start of ME process for each MB, the search pixels from the entire SW are loaded into the internal memory of the ME hardware which consists of 8 RAMs. The CB pixels are also loaded into the register array for storing the current pixels. In the first step of ME, the center of search is chosen as the origin and the search pixels required for computing the matching criteria at the center are loaded into the search register array from 8 RAMs. The search register array then provides proper search pixels to all the PEs. All the PEs start the evaluation of the matching criterion (NNMP) for all the 9 search locations simultaneously. The computed values of the NNMPs are sent to the comparator. The comparator finds the minimum NNMP value as well as the corresponding location and sends it to the process control unit. The process control unit controls the entire process of ME by sending proper control signals to all the hardware modules. The internal structure of each of the modules of the proposed hardware is provided in the following subsections.

5.4.1 Processing Element

The architecture of the PE is shown in Fig. 5.5. The PE possesses two input ports namely, C and S for reading two 16-bit vectors from the CB and the SW respectively. There are two 8-bit XOR arrays in the PE. One of the XOR arrays operates on the eight most significant bits of the 16-bit vectors C and S, and the other operates on the remaining eight least significant bits. The number of ones as a result of the XOR operation is obtained by using two look-up tables (LUTs) with 28 entries. The outputs of the LUTs are then applied to a 4-bit adder. The output of the adder is then applied at the input of the accumulator. The output of the accumulator provides the value of the NNMP for a particular location after 16 clock cycles for a MB of size 16×16.

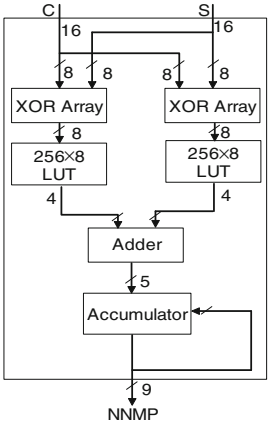


Fig. 5.5 PE architecture for 1-BT FBS ME

5.4.2 Memory Interleaving

In order to provide the required search pixels to all the PEs simultaneously, memory interleaving technique similar to [19] is exploited. The search pixels from the SW are interleaved among 8 RAMs (indexed from 0 to 7). For a single MB of size 16 × 16 pixels and for a search range of [−16, 16] pixels, the size of the SW will be 48 × 48 pixels. Here, since each pixel is represented by one bit, the actual size of the SW would be 48 × 48 bits. The distribution of the search pixels among these 8 RAMs is depicted in Fig. 5.6. All the pixels from the first location of SW are stored into the first location (that is the address 0) of the RAM0. Similarly, all the pixels from the second location of the SW are stored into the first location of the RAM1.

This process is repeated for all the first eight locations of the SW (i.e. for the locations 0 to 7). After the eighth location, the search pixels from the next location (i.e. address 8) are stored into the second location of the RAM0 (i.e. address 1). This process is repeated for the entire SW. This whole process is depicted in Fig. 5.6b. Using the above memory organization, it becomes very easy for one to locate the particular RAM and the location in that RAM where the search pixels from the SW with a given location are stored. For a given location in the SW, the last three bits of the location determine the particular RAM, whereas the first three bits of the location starting from the MSB determine the location of that RAM at which the search pixels with the given location are stored. For example, for the location 7 (000111) of the SW, the first three bits starting from the MSB (i.e. 000) determine the address of the

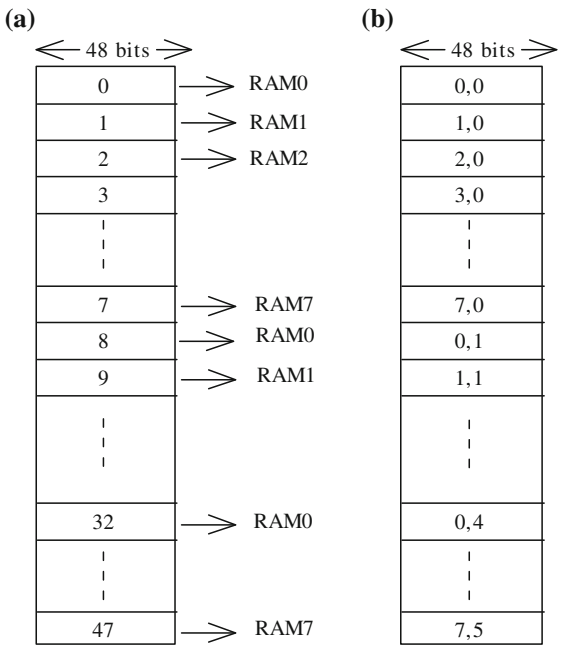


Fig. 5.6 Memory interleaving for parallel access of the search pixels. **a** Search block for MB of size 16 × 16 and search range of [−16, 16]. **b** Modified memory arrangement with the first index indicating the RAM number and the second indicating the location of 48-bit word stored in the RAM

RAM and the last three bits (i.e. 111) determine the particular RAM, i.e. the search pixels from location '7' of the SW are stored at an address 0 of the RAM7. Similarly, the search pixels from the location '47' of the SW are stored at address 5 of RAM7.

5.4.3 Register Array for the Current Block Pixels

The pixels from the CB are read only once from the external memory and are stored into an array of sixteen 16-bit registers. The current pixels are then provided to the proper PE by the register array, and no further external memory access is required. In this way, the proposed architecture significantly reduces the number of external memory accesses for the current pixels.

5.4.4 Search Register Array

In order to provide search pixels to all the PEs simultaneously, an array of five 48-bit registers are used. The data sharing and the data reuse techniques presented in Sect. 1.3 are performed by this unit. It reduces the number of external memory accesses to a great extent. At the start of each search, the data are loaded into the register array from the RAMs independently and at the same time. From the next step onwards, no external memory read operations are required for the registers R0–R3 as shown in Fig. 5.7.

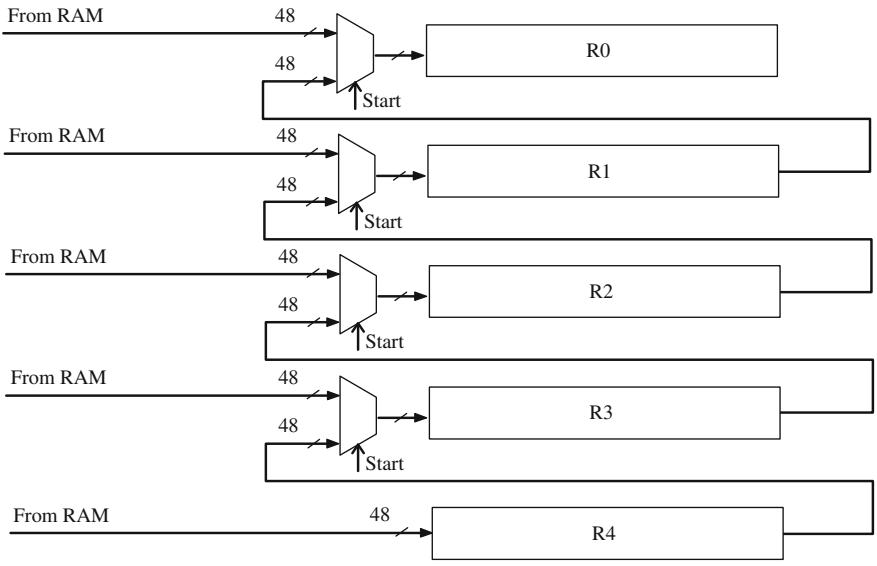


Fig. 5.7 Register array for search pixels

Register R0 is now connected to register R1 via one multiplexer and register R1 is connected to register R2 via another multiplexer. This is the case for all the registers except for the last register R4, which is always connected with one of the RAMs.

The register R0 always provides the search pixels required for the computation of the NNMP at location 0 (vide Fig. 5.2). Computation of NNMP at location 0 is done by PE0. Therefore, register R0 is connected to PE0 via one 16-bit data selector. In a similar way, R1 supplies the search pixels required for the computation of NNMP at the locations 1 and 2. Therefore, register R1 is connected to the processing elements PE1 and PE2 via two 16-bit data selectors. Register R2 is connected to PE3, PE4 and PE5 via three 16-bit data selectors. In a similar way, register R3 is connected to PE6 and PE7, while register R4 is connected to PE8 via data selectors.

The data selector provides the proper data to the proper PE from the register array. It accepts one 48-bit data at its input from the register and provides one 16-bit output to the PE that is connected to it. It selects the proper data depending upon the location of the search center. In particular, if the search center is located at the pixel coordinate (4, 4), then one of the data selectors that is attached to register R1 will accept the entire row of 48-bit search data starting from (3, 0) to (3, 47) from register R1, and will provide one 16-bit search data starting from (3, 3) to (3, 18) to PE1. Another data selector will provide 16-bit data starting from (3, 5) to (3, 20) to PE2. It takes one clock cycle to load the search pixels from the RAMs into the register array and thus it produces a delay of one clock cycle.

5.4.5 Comparator Unit

The outputs of all the PEs are connected to a comparator. The comparator selects the best MV from the computed values of the matching criteria. It generates the coordinates of the best matching point. It also generates a number ranging from 0 to 8 indicating the location where the best match is found. For example, with reference to Fig. 5.2, if the best match were found at the center of the search, then it will generate the number 4.

5.4.6 Process Control Unit

This is the most critical part of the architecture. There are two parts in this unit. One is the pattern analyzer and the other is the control sequence generator. The pattern analyzer unit takes the decision regarding whether to perform LDSP or SDSP in the next step of the search depending upon the location of the best match as obtained from the comparator unit. It also generates the center of the new search and the new points where the matching criteria are to be evaluated.

The control sequence generator generates the control signals to control the register array for the current block pixels and the search register. It generates the control

signals required to coordinate among the different hardware units to complete the search smoothly. It also generates the termination signal indicating that the search process is complete, and the MVs are ready to be read from the MV selector.

5.5 Proposed Fast Binary ME Architecture for Variable Block Size

The DS based binary ME architecture with Variable Block Size (VBS) is similar to the FBS ME architecture shown in Fig. 5.4. The VBS motion estimation architecture computes the NNMP values for all the 41 partitions of a MB. The partition of a given MB into 41 blocks is shown in Fig. 5.8. However, for VBS ME architecture, an array of 4 PEs is used instead of a single PE as was used for FBS ME architecture. The architecture of a single PE has been shown in Fig. 5.9. There are two read ports S and C for reading the row of pixels from the SW and the CB respectively. These are then applied at the inputs of the XOR array and the number of 1s as a result of XOR operation is counted by a LUT. The LUT has 16 entries. The output of the LUT is then applied to an accumulator. The final output of the accumulator provides the NNMP value for a particular location.

The detailed structure of PE Array 0 has been depicted in Fig. 5.10. PE Array 0 consists of four PEs namely, PE00, PE01, PE02 and PE03. Each PE computes the NNMP value for a primitive block of size 4×4 in 4 clock cycles. One 4-bit counter,

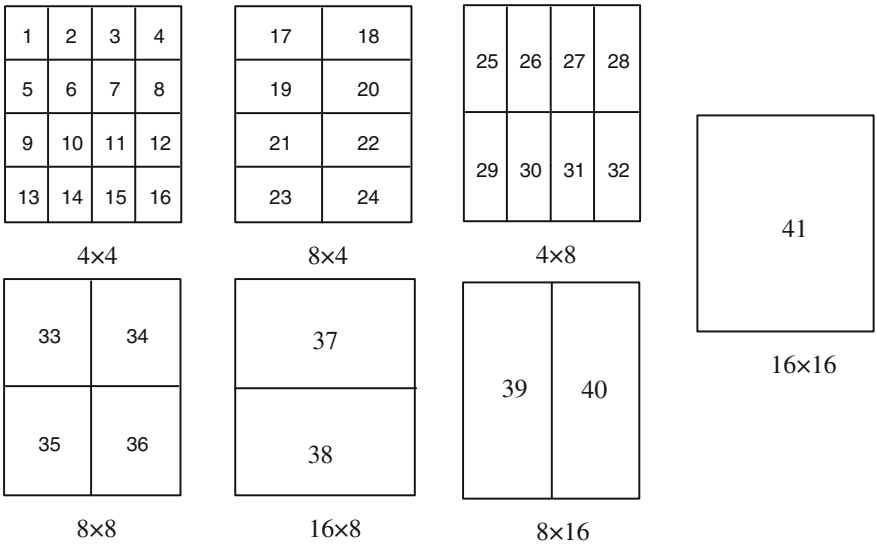


Fig. 5.8 Partition of a MB into 41 blocks

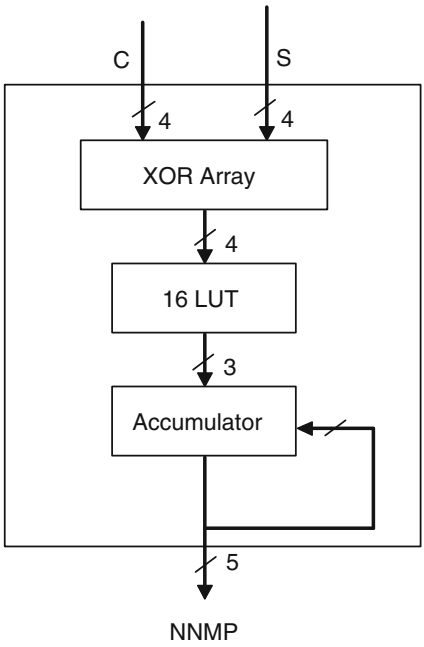


Fig. 5.9 PE architecture for 1-BT VBS ME

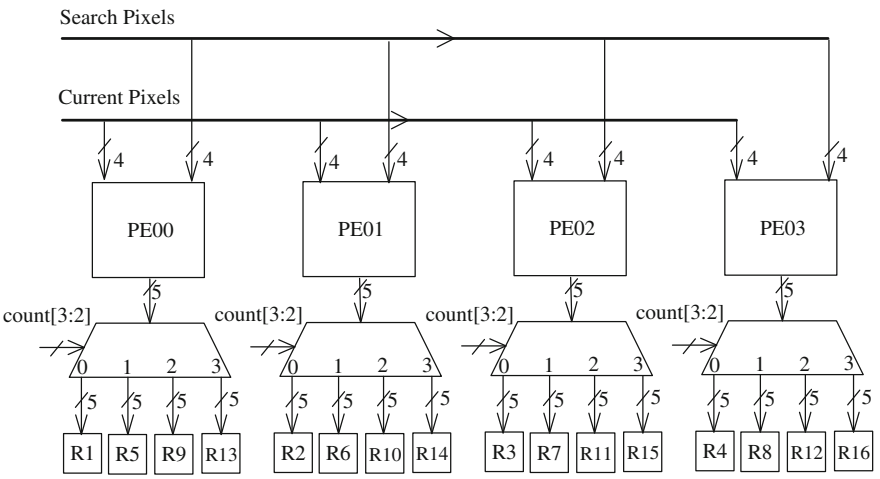


Fig. 5.10 Configuration of the PE Array 0

which starts counting from the start of the operation (i.e. from time t = 0), is used to keep track of the entire process. The NNMP values for sixteen 4 × 4 blocks are stored into sixteen 5-bit registers, namely R1 to R16 (vide Fig. 5.10). The NNMP values for sixteen 4 × 4 blocks are directed to the appropriate registers by four 1 to 4 demultiplexers. The first two bits starting from the MSB of the counter are used as the control signal for the demultiplexers. Since each PE can compute the NNMP value of a 44 block in 4 clock cycles, the four PEs, namely PE00, PE01, PE02 and PE03 work in parallel to produce the NNMP values for the blocks 1, 2, 3 and 4 (vide Fig. 5.8) respectively at time t = 3. At time t = 3, the counter will read count = 0011 and count [3:2] = 00, so the output of the PE00 will be stored in the register R1. Similarly, the outputs of the other three PEs, namely PE01, PE02 and PE03 will be stored in the registers R2, R3 and R4 respectively.

In this way, all the sixteen NNMP values for sixteen 4 × 4 blocks are obtained and are stored in the registers R1 to R16. The NNMP values for other block sizes are obtained by adding the NNMP values of the primitive 4 × 4 blocks. For example, the NNMP values of the two 4 × 4 blocks are added to obtain the NNMP value for a 4 × 8 or an 8 × 4 block. The NNMP values for the 4 × 8 and 8 × 4 blocks are stored into a pipeline register [9]. The NNMP values of the two 8 × 4 blocks are added to obtain the NNMP value of one 8 × 8 block. The NNMP value of the two 8 × 8 blocks are added to obtain the NNMP value for one 16 × 8 or 8 × 16 block, which are also stored into another pipeline register. The NNMP values of two 8 × 16 blocks are finally added to obtain the NNMP value for a single MB. The merging process introduces a delay of two clock cycles for two stage pipelining. The 41 NNMP values computed for a MB are finally sent to a comparator and a MV selector, which determine the minimum NNMP value and the corresponding MVs for each MB partition.

5.6 Results

5.6.1 Performance of the Proposed Fast 1-BT Based ME

The PSNR difference for the proposed method of ME has been compared with the conventional FS based binary ME. The results have been shown in Table 5.3. It can be observed from Table 5.3 that the proposed method of ME can provide a very small PSNR degradation compared to the FS based binary ME. For a video sequence with low motion like Hall Monitor, the PSNR drop is below 0.1 dB. However, the PSNR drop increases slightly for the sequences with complex motion like Foreman and Coastguard. The FS based binary ME has been compared with the proposed method of ME in terms of the average number of steps required to process a MB, and the result of the comparison has been provided in Table 5.4. Based on the observations made on Tables 5.3 and 5.4, it can be concluded that the computational complexity of the proposed method is much lower than the FS based ME. At the same time, the degradation in PSNR is also less than 0.5 dB for all the video sequences tested.

Table 5.3 Average PSNR drop (dB) against conventional full search based binary ME

Foreman	Hall Monitor	Football	Tennis	Coastguard
0.308	0.080	0.191	0.209	0.478

Table 5.4 Average number of search points per MV generation

Video sequence	Average no. of search steps
Foreman	6.85
Hall Monitor	3.76
Football	4.92
Tennis	5.62
Coastguard	3.98

The number of search points for TSS and FS are fixed, namely 25 and 1,089 respectively for a search range of [−16, 16]

5.6.2 Implementation Results

The proposed DS based binary ME architectures for FBS and VBS are described in Verilog HDL. In order to make a fair comparison with the architectures reported in [9], the proposed architectures are also implemented on the same FPGA as was done in [9]. The proposed fast binary FBS ME architecture consumes 731 slices (1,103 LUTs). The on-chip memory of the proposed architecture is 2,304 bits for storing the SW for a single MB. These search pixels are stored into 6 locations of 8 RAMs. The proposed VBS ME architecture consumes 1,014 slices (1,576 LUTs).

The proposed VBS ME architecture involves a latency of 22 clock cycles: one clock cycle for loading the search pixels from the RAMs into the register array for search pixels, 16 clock cycles required for the computation of the NNMP, two clock cycles for the merging process, one clock cycle for the comparison and one clock cycle each required for the purpose of the pattern generation and the control signal generation. Therefore, each of the steps of DS requires 22 clock cycles. Based on our simulation results (vide Table 5.4), it is found that when applying DS on 1-BT frames, a maximum of seven search steps are required for processing a single MB. Thus, a total 154 (22×7) clock cycles are required to process a single MB.

Comparison of the proposed architecture with various other architectures is presented in Table 5.5. It then follows from Table 5.5 that the proposed architecture involves the least number of clock cycles required for processing a single MB. One major advantage of having the least number of clock cycles requirement for processing one MB is that the clock frequency required to process a video sequence with a given frame size, and the frame rate is substantially reduced compared to other architectures. The reduction of the minimum clock frequency requirement has a direct impact on reducing the overall power consumption for ME hardware [20]. The clock frequencies required for the proposed VBS ME architecture and its counterparts [9, 10] for processing different video sequences with different frame sizes

Table 5.5 Comparison with other 1-BT based ME architectures

	Proposed (FBS)	Proposed (VBS)	[9] (FBS)	[9] (VBS)	[10]
No. of PEs/MB	9	36 (9 × 4)	256	256	16
On-chip memory	2,304	2,304	4,608	4,608	24,064
Supported block sizes	16 × 16	4 × 4 to 16 × 16	16 × 16	4 × 4 to 16 × 16	16 × 16
Search range	[−16, 16]	[−16, 16]	[−16, 16]	[−16, 16]	[−16, 15]
No. of clock cycles/MB	140	154	282	282	1,039
Area	731 slices (1103 LUTs)	1,014 slices (1576 LUTs)	4,758 slices (7280 LUTs)	6,782 slices (8702 LUTs)	944 slices (1467 LUTs)
Maximum freq. (MHz)	135	129	115	113	127

Table 5.6 Required clock frequencies of the proposed architecture and other 1-BT MT architectures for different video sequences

Frame size		CIF (352 × 288)	SDTV (1,280 × 720)	HD for PCs (1,280 × 720)	HD for TVs (1,920 × 1,080)
Frame rate (fps)		30	30	60	60
Clock rate (MHz)	Proposed	1.82	16.63	33.26	74.84
	Akin [9]	3.36	30.46	60.91	137
	Celebi [10]	12.34	112.22	224.41	505

and rates have been provided in Table 5.6. It can be observed from Table 5.6 that for the proposed architecture, a clock frequency of 16.63 MHz is sufficient to perform VBS ME for a video sequence with Standard Definition television (SDTV) frame size (1,280 × 720) and a frame rate of 30 fps. On the other hand, the required clock frequencies are 30.46 and 112.22 MHz for the architectures [9, 10] respectively for processing the same video sequence. Accordingly, the power consumption for processing the SDTV frame is only 59 mW for the proposed architecture, whereas the power consumptions are 83 and 96 mW for the architectures [9, 10] respectively.

5.7 Conclusions

In the present chapter, low power ME architectures have been developed for implementing DS on 1-BT frames with FBS and VBS support. As compared with other 1-BT based ME architectures, the proposed architectures involve the lowest latency.

The clock frequency of the proposed ME architectures therefore can be effectively reduced for low power designs. In particular, the required clock frequency for the proposed VBS ME architecture for processing SDTV frames (1,280 × 720 @ 30 fps) is 16.63 MHz. The resulting power dissipation is only 59 mW, which may be an attractive solution for portable consumer video applications typically operated by battery power. On the other hand, the frame size and the frame rate supported by the proposed architectures can also be extended subject to a clock frequency constraint. For the proposed VBS ME architecture, the required clock frequency for processing High definition (HD) resolution for TVs (1,920 × 1,080 @ 60 fps) is 74.84 MHz. The maximum operating frequencies of the proposed architectures for FPGA implementation are found to be 135 and 129 MHz for FBS and VBS respectively. The proposed architectures are therefore deemed suitable for designing consumer electronic products that require real time video processing or compression at affordable prices.

References

1. He, Z.L., Tsui, C.Y., Chan, K.K., Liou, M.L.: Low-power VLSI design for motion estimation using adaptive pixel truncation. IEEE Trans. Circuits Syst. Video Technol. **10**(5), 669–678 (2000)
2. Hsieh, C.H., Lin, T.P.: VLSI architecture for block-matching motion estimation algorithm. IEEE Trans. Circuits Syst. Video Technol. **2**(2), 169–175 (1992)
3. Gharavi, H., Mills, M.: Blockmatching motion estimation algorithms-new results. IEEE Trans. Circuits Syst. **37**(5), 649–665 (1990)
4. Natarajan, B., Bhaskaran, V., Konstantinides, K.: Low-complexity block-based motion estimation via one-bit transforms. IEEE Trans. Circuits Syst. Video Technol. **7**(3), 702–706 (1997)
5. Ertürk, S.: Multiplication-free one-bit transform for low-complexity block-based motion estimation. IEEE Signal Process. Lett. **14**(2), 109–112 (2007)
6. Lee, H., Jeong, J.: Early termination scheme for binary block motion estimation. IEEE Trans. Consum. Electron. **53**(4), 1682–1686 (2007)
7. Ertürk, A., Ertürk, S.: Two-bit transform for binary block motion estimation. IEEE Trans. Circuits Syst. Video Technol. **15**(7), 938–946 (2005)
8. Urhan, O., Ertürk, S.: Constrained one-bit transform for low complexity block motion estimation. IEEE Trans. Circuits Syst. Video Technol. **17**(4), 478–482 (2007)
9. Akin, A., Dogan, Y., Hamzaoglu, I.: High performance hardware architectures for one bit transform based motion estimation. IEEE Trans. Consum. Electron. **55**(2), 941–949 (2009)
10. Celebi, A., Urhan, O., Hamzaoglu, I., Ertürk, S.: Efficient hardware implementation of low bit depth motion estimation algorithms. IEEE Signal Process. Lett. **16**(6), 513–516 (2009)
11. Sumit K. Chatterjee., Chakrabarti, I.: Low power VLSI architectures for one bit transformation based fast motion estimation. IEEE Trans. Consum. Electron. **56**(4), 2652–2660 (2010)
12. Zhu, S., Ma, K.K.: A new diamond search algorithm for fast block-matching motion estimation. IEEE Trans. Image Process. **9**(2), 287–290 (2000)
13. Lee, E.S., Urhan, O., Chang, T.G.: Multiplication-free one-bit transform and diamond search combination for fast binary block motion estimation. In: Proceedings 15th International Conference on Signal Processing and Communications Applications, SIU-2007
14. Chen, T.C., Chen, Y.H., Tsai, S.F., Chien, S.Y., Chen, L.G.: Fast algorithm and architecture design of low-power integer motion estimation for H.264/AVC. IEEE Trans. Circuits Syst. Video Technol. **17**(5), 568–577 (2007)

15. Ding, D., Yao, S., Yu, L.: Memory bandwidth efficient hardware architecture for AVS encoder. IEEE Trans. Consum. Electron. **54**(2), 675–680 (2008)
16. Wei, C., Hui, H., Jiarong, T., Hao, M.: A high-performance reconfigurable VLSI architecture for VBSME in H.264. IEEE Trans. Consum. Electron. **54**(3), 1338–1345 (2008)
17. Parlak, M., Hamzaoglu, I.: Low power H.264 deblocking filter hardware implementations. In: Proceedings Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2007)
18. Kuhn, P.: Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation, 1st edn, pp. 120–128. Kluwer Academic Press (1999)
19. Jong, H.M., Chen, L.G., Chiueh, T.D.: Parallel architectures for 3-step hierarchical search block-matching algorithm. IEEE Trans. Circuits Syst. Video Technol. **4**(4), 407–416 (1994)
20. Ou, C.M., Le, C.F., Hwang, W.J.: An efficient VLSI architecture for H.264 variable block size motion estimation. IEEE Trans. Consum. Electron. **51**(4), 1291–1299 (2005)

Chapter 6
Efficient Pixel Truncation Algorithm and Architecture

The goal of this chapter is to introduce a new block matching algorithm, namely the Fast Two Stage Search (F2SS) algorithm and its VLSI architecture for performing low power variable block size Motion Estimation (ME) based on pixel truncation. The chapter starts with a brief discussion on ME methods which adopt the pixel truncation approach. The proposed F2SS algorithm has been presented in Sect. 6.2. Section 6.2 presents the architecture designed for implementing the proposed F2SS algorithm. The simulation and the synthesis results of the proposed algorithm and the corresponding architecture are presented in the subsequent section. The conclusions are finally presented in the last section.

6.1 Introduction

Motion Estimation (ME) is a process widely used to compress motion pictures to effectively remove temporal redundancies. Block Matching Algorithm (BMA) is the preferred type of ME due to its simplicity and performance efficiency [1–3]. In BMA, for each block in the current frame (current block), the best matched block in a previously available frame (the reference frame) is searched within a given search area. The Motion Vector (MV) is defined as the displacement between the current block and the best matched block in the reference frame. Although ME based on Full Search (FS) algorithm provides the optimum solution for obtaining high compression ratio, they suffer from high hardware costs and computational power requirements. In modern video coding standards such as MPEG-4/H.264, a special feature called Variable Block Size Motion Estimation (VBSME) has been included to improve coding efficiency.

Due to the increasing need for advanced video coding methods, many portable electronic devices such as mobile phones and camcorders typically use MPEG-4 or H.264 based techniques for video compression. These mobile devices are operated by batteries and thus have limited processing power. To suit these conditions, many fast ME algorithms such as four step search [4], diamond search [5], and adaptive

I. Chakrabarti et al., *Motion Estimation for Video Coding*,
Studies in Computational Intelligence 590, DOI 10.1007/978-3-319-14376-7_6

rood pattern search [6] have been proposed. These fast search algorithms reduce computational complexity of the FS algorithm without compromising Peak Signal-to-Noise Ratio (PSNR). In another approach, pixel resolution is reduced from 8 bits to fewer bits. One-Bit Transform (1-BT) based MEs have been proposed in [7] and [8], where each pixel is transformed to a one-bit representation. In [9] the bit-depths of the pixels are truncated. It is also shown in [9] that on an average, four bits can be truncated without adversely affecting the quality of the reconstructed frame. There are several ME architectures based on 1-BT. The first implementation of 1-BT based VBSME was presented in [10]. The first implementation of ME based on 1-BT and multiple reference frames was proposed in [11]. The implementation of fast 1-BT based ME was proposed in [12] by combining 1-BT based ME with diamond search algorithm. Recently, an efficient algorithm and its architecture based on pixel truncation was presented in [13] for performing VBSME.

In the present chapter, an extension of the work presented in [13] is reported. It is shown that the present algorithm is faster and more efficient than that presented in [13]. The resulting architecture also consumes less power than the algorithm described in [13]. The chapter is organized as follows. The new algorithm for performing ME is presented in the following section. Section 6.3 presents the proposed architecture. The simulation and the synthesis results are presented in Sect. 6.4. Conclusions are finally drawn in Sect. 6.5.

6.2 Proposed Fast Two Stage Search Based Motion Estimation Algorithm

In the present algorithm, ME is performed in two stages: in the first stage, ME is performed on truncated pixels, and in the second stage, it is performed on the pixels with full resolution for refining the MV obtained from the first stage. It has been shown in [13] that optimum results may be obtained by taking the two Most Significant Bits (MSBs) of the pixel and by using the Difference Pixel Count (DPC) as the matching criterion. Therefore, in the present ME algorithm, the two MSBs are taken and DPC is used as the matching criterion as proposed in [13]. Thus, the first stage of the proposed algorithm is essentially the same as the first stage of the algorithm [13]. For the second stage of search, an algorithm is proposed based on experiments performed on different benchmark video sequences. Based on these experimental results, it is found that there is a high possibility that if the MV for the first stage points in a particular direction, the MV for the second stage will also have similar characteristics. Substantial savings in computational time can therefore be achieved by restricting the second stage of search in the direction pointed by the MV obtained from first stage of search.

It is also found that there is a high probability that the search for a large MV in a small search pattern with closely spaced search points may be trapped in some local minimum [5]. On the other hand, for detecting small motions, a small search pattern

with closely spaced search points is more suitable than a large search pattern with widely spaced search points [5]. From the above discussions, it may be concluded that the computational time may be saved further by selecting the search pattern for the second stage of the search in accordance with the magnitude of the MV obtained from the first stage. Based on these factors, a novel scheme for second stage MV refinement has been proposed. The details of the algorithm are given below. In this algorithm, an initial search pattern is selected based on both the magnitude and the direction of the MV obtained from the first stage of search. The search pattern is chosen in such a way that it is able to detect both large and small or zero MVs efficiently. It is also shown that, for most real-world video sequences, MV distributions in the horizontal and the vertical directions are higher than in other directions [6]. Based on these observations, a rood-like search pattern is selected as the initial search pattern similar to [6].

The proposed Fast Two-Stage Search (F2SS) ME algorithm can be represented pictorially as shown in Fig. 6.1. In Fig. 6.1, it is assumed that the MV obtained from the first stage points to the location $(-3, +2)$, represented as a solid sphere. A rood-like search pattern is then created around this location with four search points at the four vertices of the rood pattern, shown as transparent squares in Fig. 6.1. The distance S between the center and the search points is selected as $S = \max\{|MV_x|, |MV_y|\}$, where $\{|MV_x| \text{ and } |MV_y|\}$ are the vertical and the horizontal MV components respectively. In the second stage of the proposed algorithm, the search is performed at all five points (including the center) after selecting the initial search pattern. The point

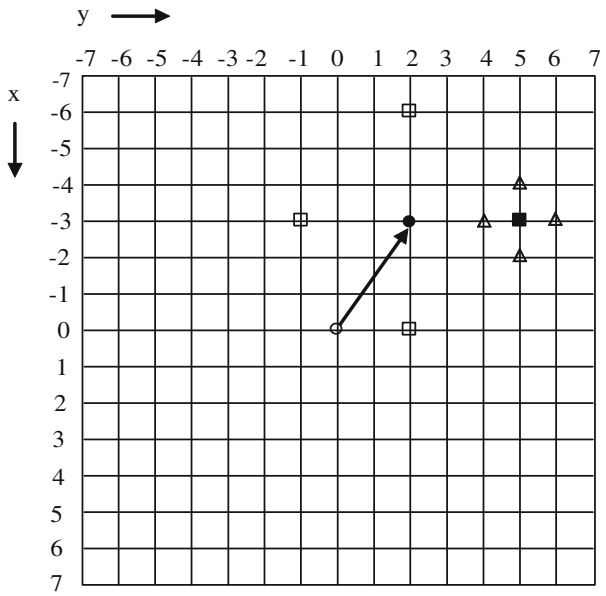


Fig. 6.1 The search strategy for the proposed F2SS algorithm

corresponding to the minimum Sum of Absolute Differences (SAD) value (SADmin) is chosen as the new center of search and is indicated by a solid square in Fig. 6.1. If SADmin is found at the center of the rood pattern, then the search is stopped midway; else, the search pattern is changed to the Small Diamond Search Pattern (SDSP) around SADmin, which is the same search pattern that was used in [5]. The location of the search points for subsequent search steps are indicated by the triangles. The above process is repeated until SADmin is found at the center of the SDSP.

The present algorithm is motivated by the Adaptive Rood Pattern Search Algorithm (ARPS) originally proposed in [6]. However, the algorithm presented here is different from ARPS in the following aspects. (1) In ARPS, the initial search pattern is selected based on the MVs of the neighboring blocks. In this one however, the initial search pattern for the second stage is selected from the MV obtained from the first stage. (2) Since the second stage of refinement is performed around the MV obtained from the first stage in the present case, no further checking is required for the Zero-Motion Prejudgment (ZMP). However, for ARPS, extra checks are required for ZMP resulting in increased complexity. In the algorithm proposed in [13], FS is performed during the second stage of the search. Additionally, in the second stage, the search area is reduced to quarter of the first search area. This results in a substantial drop in the PSNR value of the reconstructed image for the sequences with complex motions like Foreman. On the other hand, for the present algorithm, the same search area is considered in both stages.

The main advantage of the present algorithm over [13] is its ability to perform faster MEs. This is because, if the best match is found at the center at the first step of the second stage of ME, then it does not waste any time in further search. Also, the proposed algorithm saves computational time for detecting large MVs by directly jumping to the neighborhood of the most likely location of the MV. Subsequently, SDSP is used in the present method in contrast to the method in [13] that is based on the FS algorithm, which consumes considerable time in checking all possible locations. The proposed algorithm is summarized in the following subsection.

6.2.1 Summary of the Proposed Fast Two Stage Search Algorithm

First Stage:
1. **Perform** FS based ME on truncated pixels.
2. **Obtain** the location that is indicated by the MV.

Second Stage:
1. Construct a rood-like search pattern around the location obtained from the First Stage with four search points located at the four vertices of the rood pattern with $S = \max\{|MV_x|, |MV_y|\}$.
2. Compute the SAD values at the four search locations and at the center and also obtain SADmin.

if (SADmin is found at the center) **then**
MV = $\{|MV_x|, |MV_y|\}$;
stop;
else
go to 3;
3. Change the search pattern to SDSP around the point corresponding to SADmin and compute the SAD values at all the search locations to obtain the new value of SADmin.
if (SADmin is found at the center) **then**
read MV corresponding to SADmin;
stop;
else
go to 3;

6.3 Architecture for the Proposed Fast Two Stage Search Algorithm

In order to reduce computational complexity, many ME architectures based on pixel truncation have been proposed [9, 13]. In these architectures, the entire 8-bit data are accessed, and only a part of the data is used to evaluate the matching criteria. Although computational complexity is reduced substantially by this approach, there is no reduction in the memory bandwidth requirement. In a recently reported architecture [13], a new memory design has been presented, in which different number of bit-planes can be accessed at different stages of ME from the same memory module. The proposed scheme is attractive but requires additional hardware to transpose and realign the pixels during ME.

6.3.1 Memory Management for the Proposed F2SS Algorithm

In the present memory management scheme, two on-chip memories namely, RAM0 and RAM1 are used instead of a single on-chip memory. The first two MSBs of the reference pixels are stored in RAM0, and the remaining 6 bits are stored in RAM1 as shown in Fig. 6.2. Here, $R_{0,0}$[7:6] represents the first two MSBs, while $R_{0,0}$[5:0] represents the remaining 6 bits of the reference pixels from the Search Window (SW) from the location (0,0). This process is repeated for the entire SW, and all the reference pixels are arranged in two RAMs as depicted in Fig. 6.2. Similarly, the current pixels are also organized in two RAMs—CBRAM0 and CBRAM1. The ME architecture with this new memory organization is shown in Fig. 6.3. In the first stage of ME, the most significant 2 bits of the reference pixels can be directly accessed from RAM0, and two MSBs of the current pixels can be accessed from CBRAM0.

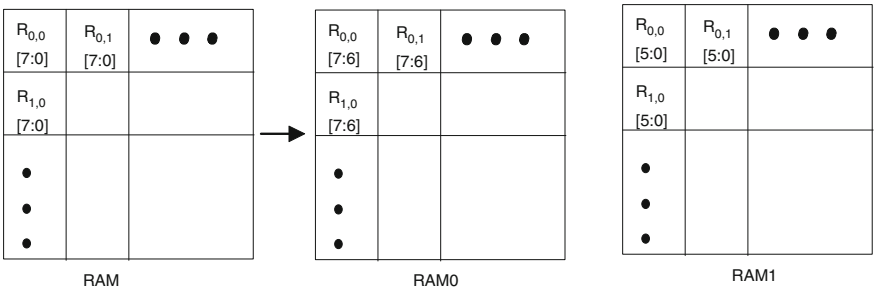


Fig. 6.2 The proposed memory management scheme

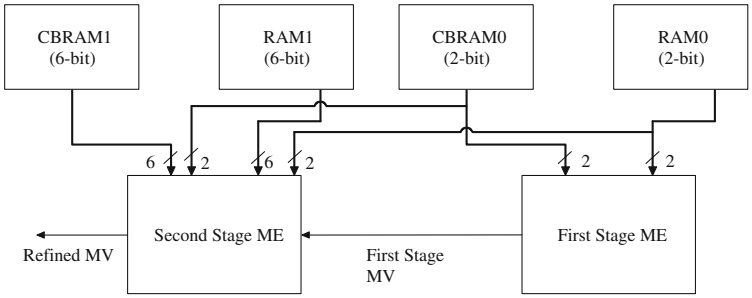


Fig. 6.3 The block diagram of an ME architecture with the proposed memory management scheme

After the first stage of ME, the second stage of MV refinement is performed by a refinement unit shown as the block "Second Stage ME" in Fig. 6.3, for which both RAMs are used.

6.3.2 Proposed Architecture for the First Stage of ME

As previously mentioned, in the first stage of the present F2SS algorithm, ME is performed by taking the two MSBs of the pixel and by using the DPC as the matching criterion. For a block of size N × N and for a search range [−p, p−1], the DPC at any location (m, n) can be found as [14]:

$$DPC(m,n) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \bar{\delta}[\hat{C}(i,j), \hat{R}(i+m, j+n)] \tag{6.1}$$

Here, $\hat{C}(i, j)$ and $\hat{R}(i+m, j+n)$ represent bit truncated values for the pixels from the CB and the SW respectively. In Eq. 6.1, $\bar{\delta}(x, y)$ represents the standard delta

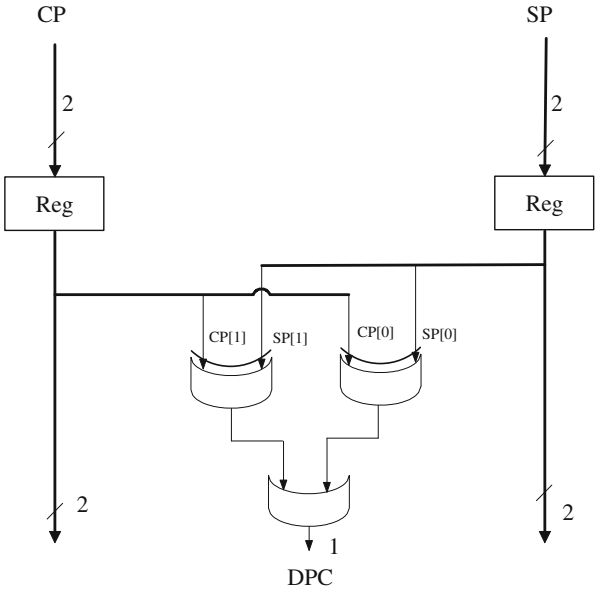


Fig. 6.4 The architecture for the DPC based PE. Here, CP and SP represent 2 MSBs of the current and the search pixels respectively

function, for which $\bar{\delta}(x, y) = 0$ if (x = y); else its value is 1. The implementation of DPC on hardware is much simpler than conventional SAD. The architecture for the DPC based Processing Element (PE) is shown in Fig. 6.4.

The architecture performing the first stage of ME is shown in Fig. 6.5, which is based on SAD tree [15], and is similar to that reported in [13]. The architecture consists of the Current Block (CB) memory, a search register array; a 2-D array of 256 DPC based PEs, a 2-D adder tree, a comparator, and the final MV selector. However, owing to the proposed memory management scheme, for the first stage of ME, the reference and the current pixels are accessed from RAM0 and CBRAM0 respectively. The reference pixels are stored in the register array for data reuse, which in turn reduces the required memory bandwidth. For a block of size N × N, a total of N × N current and reference pixels are loaded into PEs in every clock cycle. At the same time, N reference pixels that belong to the same row of the SW are also loaded into the register array to up date the reference pixels [15]. The reference pixels are propagated in the vertical direction rowby- row in the register array.

After an initial latency, the PEs generate 256 pixel differences for a given location in every clock cycle. These are then added by the adder tree to produce DPC for all the blocks. As depicted in Fig. 6.6, a Macroblock (MB) can be partitioned into 41 blocks. Thus, the adder tree generates 41 DPCs in all. The comparator and MV selector select the best MB partition and the corresponding MVs.

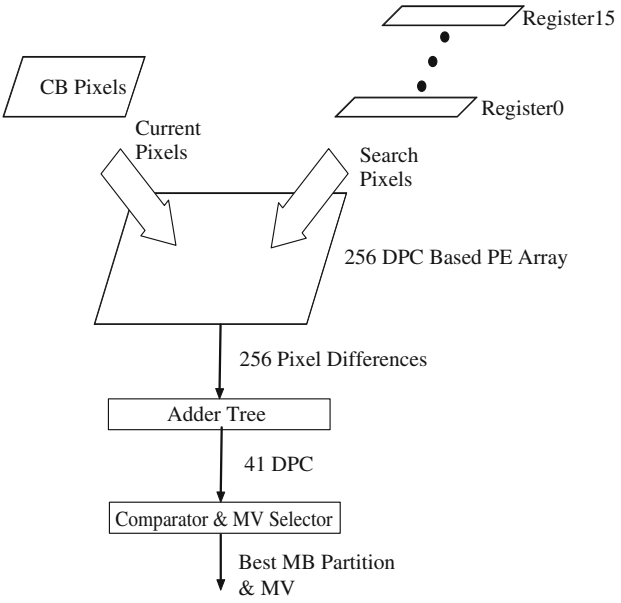


Fig. 6.5 The block diagram for architecture performing the first stage of ME

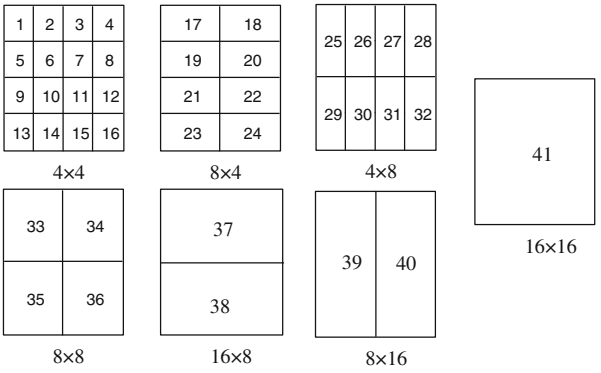


Fig. 6.6 The partition of a MB into 41 blocks

6.3.3 Proposed Architecture for the Second Stage of ME

In the second stage of the proposed F2SS algorithm, ME is performed on the pixels with full resolution, and Sum of Absolute Differences (SAD) is used as the matching criterion. For a block of size N × N, the SAD at any location (m, n) within a search range [−p, p−1] can be found as:

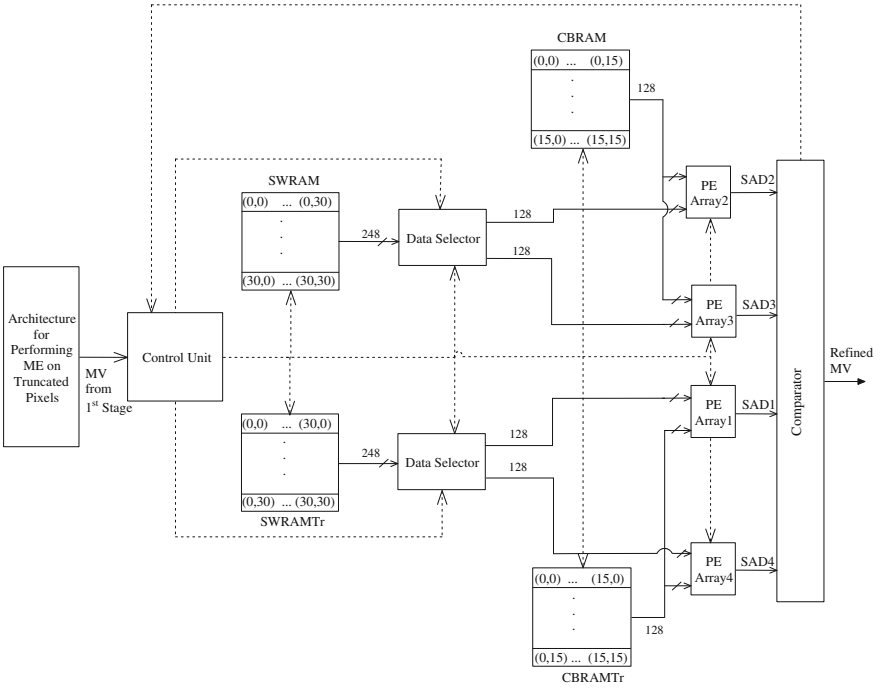


Fig. 6.7 The block diagram of the proposed ME architecture

$$SAD(m,n) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |C(i,\, j) - R(i+m,\, j+n)| \tag{6.2}$$

Here, $C(i,\, j)$ and $R(i+m,\, j+n)$ are the pixel values for the current and the reference pixels respectively.

Figure 6.7 shows the block diagram of the proposed architecture. The architecture includes two on-chip memories for storing the reference pixels from the SW, two on-chip memories for storing the current pixels from the CB, two data selectors, 4 PE arrays, one control unit, and one comparator. Each of the modules of the proposed architecture is described in detail as follows.

6.3.3.1 On-Chip Memory Unit

The proposed architecture is designed for performing VBSME for a MB of size 16×16 pixels, and for a search range of $[-8, 7]$ pixels. Therefore, the total

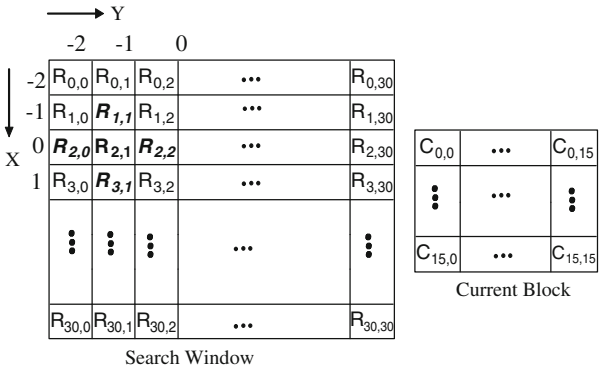


Fig. 6.8 The search window and the current block

size of the Search Window (SW) is 31 × 31 pixels. Figure 6.8 shows the CB and the corresponding SW. Let us further assume that the MV obtained from the first stage points to the location (0, −1), which corresponds to the reference pixel $R_{2,1}$. Then, according to the proposed algorithm, one is required to search at the locations (−1, −1), (0, −2), (0, 0) and (1, −1). The starting reference pixels corresponding to these locations are $R_{1,1}$, $R_{2,0}$, $R_{2,2}$ and $R_{3,1}$ respectively.

The rood-like search pattern for the proposed algorithm is shown in Fig. 6.9. Here, the search center is denoted by '0', and all the four search points around the center are denoted by the indices '1' to '4'. All the pixels from the SW are stored in two RAMs namely, SWRAM and SWRAMTr. However, owing to the proposed memory management scheme as shown in Fig. 6.2, both the RAMs consist of two smaller RAMs. In order to make the overall process faster, the entire row of pixels from the SW is written as a single word to a single address of SWRAM. As there are 31 pixels in one row of the SW, these are written as a single 248-bit (31 pixels × 8 bits) word in SWRAM. In another RAM (SWRAMTr in Fig. 6.10), the same reference pixels are stored but in transposed form, i.e. the entire first column of

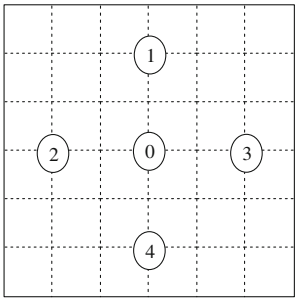


Fig. 6.9 The rood-like search pattern for the second stage of the proposed F2SS algorithm

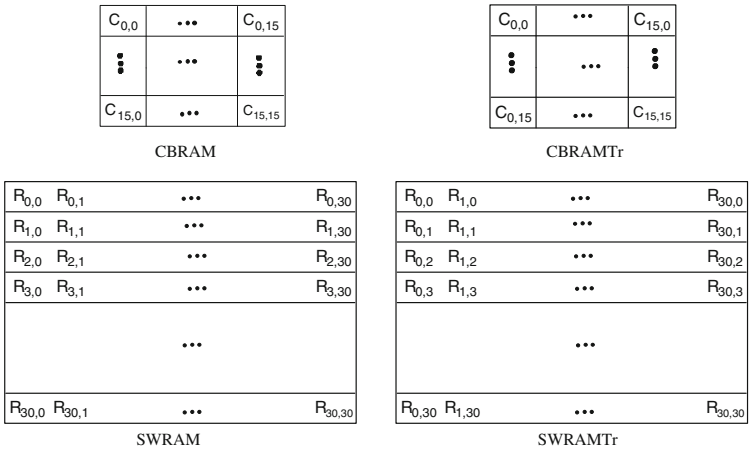


Fig. 6.10 The modified memory arrangement for the proposed architecture

pixels from the SW starting from $R_{0,0}$ to $R_{30,0}$ are written as a single 248-bit word to the address '0' of SWRAMTr. This new memory arrangement is shown in Fig. 6.10. As shown in Fig. 6.10, the current pixels are also stored in two RAMs, namely CBRAM and CBRAMTr (both of which consist of two smaller RAMs as shown in Fig. 6.2). In this case, an entire row of pixels from the CB is written as a single 128-bit (16 pixels $\times$ 8 bits) word at a single address of the CBRAM. In another RAM namely, CBRAMTr, an entire column of the pixels from the CB is stored as a single 128-bit word. It is now obvious from Figs. 6.9 and 6.10 that the entire first row of reference pixels corresponding to the search locations '2' and '3' ($R_{2,0}$ to $R_{2,15}$ and $R_{2,2}$ to $R_{2,17}$) belong to the same address of the SWRAM. Similarly, the entire first column of reference pixels corresponding to the search locations '1' and '4' ($R_{1,1}$ to $R_{16,1}$ and $R_{3,1}$ to $R_{18,1}$) belong to the same address of SWRAMTr.

6.3.3.2 Data Selector Unit

The outputs of SWRAM and SWRAMTr are connected to the data selectors. Each data selector accepts the entire 248-bit data from the RAM, and provides two 128-bit (16 pixels $\times$ 8 bits) data to the PE arrays that are connected to it. It selects the proper data depending upon the location of the search center as obtained from the control unit.

6.3.3.3 PE Array

As the second stage of the proposed F2SS algorithm involves four search locations, four PE arrays are used. In each of the PE arrays, the number of PEs is equal to

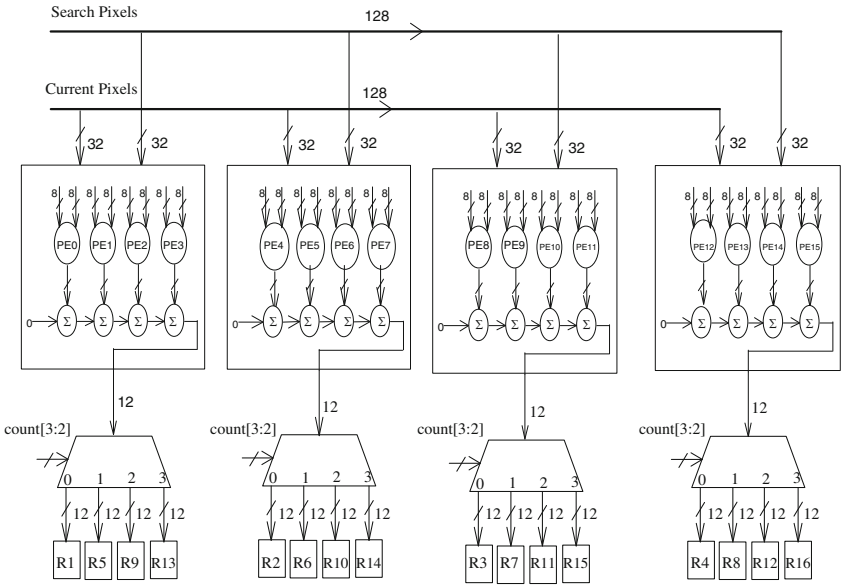
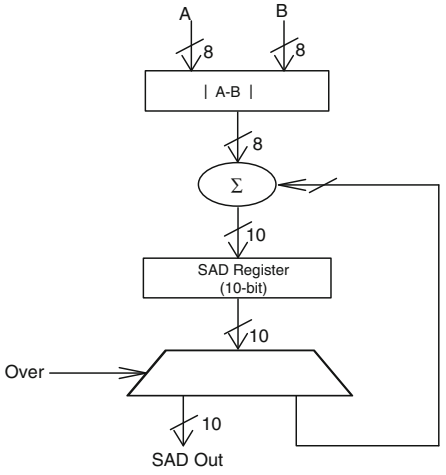


Fig. 6.11 The detailed architecture of a PE array

Fig. 6.12 The structure for a single PE



the size of the row of the CB. Moreover, as in the present example the size of the CB is taken as 16×16, there are 16 PEs in each of the PE arrays. The detailed structure of a PE array and a single PE are shown in Figs. 6.11 and 6.12 respectively. One 4-bit counter, which starts counting from the start of the operation (i.e. from time $t = 0$), is used to keep track of the entire process. The SAD values for sixteen 4×4 blocks are stored into sixteen 12-bit registers, R1 to R16 (vide Fig. 6.11). The SAD

values for sixteen 4 × 4 blocks are directed to the appropriate registers by four 1 to 4 demultiplexers. The first two bits starting from the MSB of the counter output are used as the control signal for the demultiplexers. The first group of four PEs, namely PE0, PE1, PE2 and PE3 work in parallel to produce the SAD values for block 1 (vide Fig. 6.6) in four clock cycles (or at time t = 3). At the same time, the SAD values for blocks 2, 3 and 4 are also produced by the other three groups of PEs. At time t = 3, the counter will read count = 0011 and count [3:2] = 00, and hence the SAD values for block 1 will be stored in the register R1. Similarly, the SAD values for the blocks 2, 3 and 4 will be stored in the registers R2, R3 and R4 respectively. In this way, all the sixteen SAD values for sixteen 4 × 4 blocks are obtained, and these are stored in the registers R1 to R16. The SAD values for the other block sizes are obtained by adding the SAD values of the primitive 4 × 4 blocks [10].

6.3.3.4 Comparator

All the PE arrays send the computed values of SAD to the comparator. The comparator finds the value of SADmin and the corresponding location. It then generates a number ranging from 0 to 4 depending upon the location of the SADmin and sends the same to the control unit.

6.3.3.5 Control Unit

The architecture performing ME on truncated pixels sends the MV to the control unit, which then generates the read addresses for all the memory units. Depending upon the location of the SADmin as obtained from the comparator unit, it further takes the decision on whether the subsequent steps of ME are required or not. If further steps of ME are required, then it generates the new read addresses for all the memory modules so that the proper reference pixels can be sent to the PE arrays. It further checks for the boundary conditions for all the steps and all of the search locations. If the boundary condition is reached at any step and for any particular location(s), then it sends the proper signal to disable the PE array(s), which is (are) responsible for the evaluation of SAD for that (those) location(s). It also sends an appropriate control signal to the comparator so that the maximum SAD value(s) is (are) initialized for that (those) location(s).

6.4 Results

6.4.1 Performance Analysis of the Proposed Algorithm

The performance of the proposed F2SS algorithm has been compared in terms of the quality and the computational cost with the FS algorithm. In Table 6.1, the average PSNR values as obtained for the proposed F2SS and the FS algorithms are presented

Table 6.1 The average PSNR obtained for the FS and the F2SS algorithm

Video sequence (CIF @ 30 fps)	Full search (dB)	Proposed (dB)
Foreman	34.82	34.66
Coastguard	30.03	29.86
Tennis	32.48	32.39
News	32.59	32.48
Stefan	28.42	28.03
Mother and daughter	33.86	33.72
Mobile	31.33	31.31
Hall monitor	33.45	33.38

for 16×16 block sizes and for a search range of $[-16, 15]$ pixels. All the sequences are in CIF format (352×288 pixels) and cover a wide range of motion content. In the present analysis, 100 frames for each of the video sequences are taken, and the average values for the PSNR per frame for each of the reconstructed video sequences are computed. It can be observed from Table 6.1 that the proposed ME algorithm displays performance similar to the FS algorithm for most of the video sequences with less than 0.2 dB degradation in the PSNR. However, for the sequence "Stefan", the PSNR degrades by 0.39 dB.

The performance of the proposed algorithm has also been compared with a similar algorithm [13]. The PSNR drop with respect to the FS algorithm as obtained by comparing the results of applying the proposed algorithm and the algorithm [13] has been presented in Table 6.2. For the proposed F2SS algorithm, 8×8 block partition is used for the first stage of search in the present analysis. Once again, from Table 6.2, it is obvious that the proposed algorithm provides a smaller PSNR drop when compared to the algorithm [13], except for the sequence "Stefan" for which the proposed

Table 6.2 The average PSNR drop (dB) for the F2SS and algorithm [13] with respect to the FS algorithm for QCIF frames

Video sequence (QCIF @ 30 fps)	Block size	Algorithm [13]	Proposed
Akio	16×16	0	0
	8×8	0	0
	4×4	0.05	0.03
Mobile	16×16	0	0
	8×8	0. 02	0.02
	4×4	0.14	0.09
Foreman	16×16	0.07	0.03
	8×8	0.19	0.16
	4×4	0.44	0.33
Stefan	16×16	0.03	0.05
	8×8	0.11	0.18
	4×4	0.27	0.25

algorithm results in a slight increase in the average PSNR drop. This may be ascribed to the fact that this sequence contains more movements in the directions other than vertical and horizontal directions which the proposed algorithm has failed to detect.

The proposed algorithm has also been tested on video sequences with CIF frame resolution and the comparison results have been shown in Table 6.3. However, it should be noted that although the hardware implementation performs matching in the search range of $[-8, 7]$ for all block sizes, a search range of $[-16, 15]$ is used for a block of dimension 16×16, a search range of $[-8, 7]$ is used for a block of dimension 8×8, while a search range of $[-4, 3]$ is used for a block of dimension 4×4, because of the available results presented in [13]. It should be further noted that these results are provided using an open loop scheme, that is, the PSNR is computed between the original frames and the motion compensated frames without using a video encoder.

Finally, it may be observed from Tables 6.2 and 6.3 that the proposed algorithm shows better performance in terms of the PSNR for most of the cases compared to the algorithm [13]. This may be attributed to the fact that in the proposed algorithm, there is no reduction in the size of the SW in the refinement stage. The average number of search steps required to process one MB can be regarded as a metric to measure the computational complexity of a given ME algorithm. In Table 6.4, the average number

Table 6.3 The average PSNR drop (dB) for the F2SS and algorithm [13] with respect to the FS algorithm for CIF frames

Video sequence (QCIF @ 30 fps)	Block size	Algorithm [13]	Proposed
Mobile	16×16	0.11	0.02
	8×8	0.20	0.13
	4×4	0.40	0.28
Foreman	16×16	0.12	0.11
	8×8	0.21	0.19
	4×4	0.39	0.31
Stefan	16×16	0.04	0.08
	8×8	0.09	0.11
	4×4	0.32	0.30

Table 6.4 The average number of steps required for second stage of the proposed F2SS algorithm

Video sequence (CIF @ 30 fps)	Number of search steps required
Foreman	2.93
Coastguard	2.85
Tennis	2.53
News	2.81
Stefan	2.92
Mother and daughter	2.03
Mobile	1.88
Hall monitor	1.09

Table 6.5 Implementation results

Technology (nm)		90
Maximum operating frequency (MHz)		129
Area (k Gates)	Processing unit	298.86
(2-input NAND gate)	Memory unit	158.98
Power (mW)		19.53 @ 100 MHz

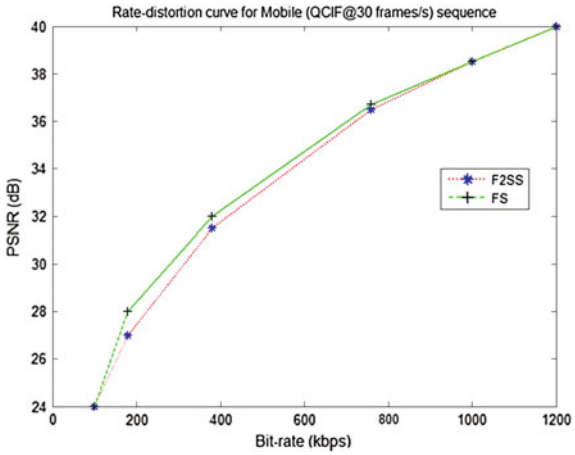


Fig. 6.13 Comparison of the rate-distortion curves for the proposed algorithm with the full-search algorithm

of search steps per MB as required by the proposed F2SS algorithm has been shown for different video sequences. It can be observed from Table 6.5 that the proposed algorithm requires at most three steps to compute the second stage of search.

In order to compare the performance of the proposed algorithm with that of the optimum FS algorithm, two video sequences namely, "Mobile" (QCIF) and "Foreman" (CIF) are selected at 30 frames per second (fps). The tests were done on the existing H.264 reference software (JM 16.0) with rate control option off and five Quantization Parameters (QP) selected as: 24, 28, 32, 36 and 40. The search range and the number of frames have been taken as [−16, 15] and 100 respectively. Here, it should be mentioned that in the present analysis, the size of the MB has been taken as 16×16 for all the cases. The test results have been illustrated in Fig. 6.13. It can be observed that the coding performance for the proposed algorithm is quite similar to that of the FS algorithm even for lower bit rates and for sequences with complex motion like "Foreman".

6.4.2 Synthesis Results and Comparison

The proposed architecture has been described in Verilog HDL and synthesized with 90 nm technology. The implementation results have been shown in Table 6.5. Before

the start of the second stage of the proposed F2SS algorithm, all the reference pixels required for performing the search are loaded into the on-chip memory unit from the off-chip memory. As mentioned earlier, the size of the SW for the proposed architecture is 31 × 31 pixels. In the proposed architecture, 31 pixels are loaded in one clock cycle. Therefore, 31 clock cycles would be required to load all the pixels from the SW into the RAMs storing the reference pixels.

In the first stage of ME, 256 clock cycles are required for low resolution search similar to the architecture described in [13]. However, in the second stage of the search procedure, the architecture involves a latency of 21 clock cycles for each of the steps as explained next. It spends sixteen clock cycles for the computation of the SADs, two clock cycles for the merging process similar to [10], one clock cycle for comparison, and one clock cycle each required for the pattern generation and control signal generation. For the proposed F2SS algorithm, if the best match is found at the center in the first step of the second stage, then no further search steps are required, and thereby the second stage of search can be completed in 52 (31 + 21) clock cycles. Thus, it can generate the MV for a 16 × 16 MB in 308 (256 + 52) clock cycles. However, based on the simulation results (vide Table 6.4), the proposed F2SS algorithm requires at most three steps to complete the second stage of the search procedure. Therefore, even in the worst case, the proposed architecture can complete the second stage of search in 63 (21 × 3) clock cycles. In addition to this, 31 clock cycles are also required for the loading of the reference pixels. Therefore, the proposed architecture can find the MV for a MB in 350 (256 + 63 + 31) clock cycles.

On the other hand, the architecture [13] always requires 500 clock cycles to process a MB. Therefore, compared to the architecture [13], the proposed architecture requires 30 % less number of clock cycles to process a MB. One major advantage of requiring fewer clock cycles for processing a MB is that the clock frequency necessary to process a video sequence with a given frame size and frame rate is substantially reduced. It can be observed that for the proposed architecture, a clock frequency of 1.04 MHz is sufficient to perform ME for a video. The proposed ME architecture has been compared with other architectures, and the results of comparison have been listed in Table 6.6. In the proposed architecture, the reference pixels are stored into two RAMs. Similarly, the current pixels are also stored in two RAMs. The memory unit is much simpler for the proposed architecture, though it requires more area to store the current and the reference pixels. The proposed architecture is based on 4 PE arrays, with 16 PEs in each array. On the other hand, for the architecture [13] which is also based on pixel truncation, 256 PEs are required. Owing to this fact, the processing unit for the proposed architecture consumes less area. Taking all of these facts into account, it is clear from the third row (gate count) of Table 6.6 that the total area for the proposed architecture is slightly on the higher side (5 % only) as compared with the architecture [13]. Since for the architectures presented in Table 6.6, different processes and supply voltages are used, the power results for all the architectures have been normalized according to the supply voltage and the dimension for the purpose of fair comparison [16]. The normalized power results for all the architectures have been shown in the last entry of Table 6.6. As mentioned earlier, since the average computational complexity is generally lower than the worst

Table 6.6 Comparison of the proposed architecture with other architectures

	[16]	[17]	[13]	Proposed
Process (nm)	180	350	130	90
Voltage (V)	1.3	3.3	1.2	1.0
Gate count (k)	131.2	23.1	436	457.5
Core size (mm^2)	3.6	7.5	2.26	2.39
Required frequency (MHz)	13.5	25	1.4	0.95
Video specification	CIF @ 30 fps	CIF @ 30 fps	QCIF @ 30 fps	QCIF @ 30 fps
Power (mW)	16.72	189	1.33	0.32
Normalized power (mW) (1.0 V, 90 nm)*	2.47	1.15	0.44	0.32

(*) Normalized power = Power $\times$ $(0.065^2/\text{Process}^2) \times (1.08^2/\text{Voltage}^2)$

case, the operating frequency can be reduced for further reduction of power consumption. This is also obvious from the last entry of Table 6.6, which reveals that the total power consumption for the proposed architecture is reduced by 27 % compared to the architecture [13], which is the best low-power video compression architecture for mobile communication available so far in the literature.

6.5 Conclusions

In this chapter, a fast block matching motion estimation algorithm, which is faster compared to a recently reported algorithm, has been proposed. The proposed algorithm is found to have relatively lower computational complexity while maintaining an acceptable image quality. The chapter also describes an appropriate architecture for implementing the proposed ME algorithm. It has been shown that, for real time encoding of QCIF videos (30 frames per second), the power consumption is only 0.32 mW with a compression performance similar to that of the full search algorithm. Therefore, the proposed architecture is particularly suitable for applications requiring low power consumption such as mobile consumer equipment.

References

1. Netravali, A.N., Robbins, J.D.: Motion compensated television coding: part-I. Bell Syst. Tech. J. **58**, 631–670 (1979)
2. Jain, J.R., Jain, A.K.: Displacement measurement and its application in interframe image coding. IEEE Trans. Commun. **29**, 1799–1808 (1981)
3. Dufaux, F., Moscheni, F.: Motion estimation techniques for digital TV: a review and a new contribution. Proc. IEEE **83**, 858–876 (1995)

4. Po, L.-M., Ma, W.-C.: A novel four step search algorithm for fast block motion estimation. IEEE Trans. Circuits Syst. Video Technol. **6**(3), 313–317 (1996)
5. Zhu, S., Ma, K.K.: A new diamond search algorithm for fast block-matching motion estimation. IEEE Trans. Image Process. **9**(2), 287–290 (2000)
6. Nie, Y., Ma, K.K.: Adaptive rood pattern search for fast block-matching motion estimation. IEEE Trans. Image Process. **11**(12), 1442–1448 (2002)
7. Natarajan, B., Bhaskaran, V., Konstantinides, K.: Low-complexity block-based motion estimation via one-bit transforms. IEEE Trans. Circuits Syst. Video Technol. **7**(3), 702–706 (1997)
8. Ertürk, S.: Multiplication-free one-bit transform for low–complexity block-based motion estimation. IEEE Signal Process. Lett. **14**(2), 109112 (2007)
9. He, Z.L., Tsui, C.Y., Chan, K.K., Liou, M.L.: Low-power VLSI design for motion estimation using adaptive pixel truncation. IEEE Trans. Circuits Syst. Video Technol. **10**(5), 669–678 (2000)
10. Akin, A., Dogan, Y., Hamzaoglu, I.: High performance hardware architectures for one bit transform based motion estimation. IEEE Trans. Consum. Electron. **55**(2), 941–949 (2009)
11. Akin, A., Sayilar, G., Hamzaoglu, I.: High performance hardware architectures for one bit transform based single and multiple reference frame motion estimation. IEEE Trans. Consum. Electron. **56**(2), 1144–1152 (2010)
12. Chatterjee, S.K., Chakrabarti, I.: Low power VLSI architectures for one bit transformation based fast motion estimation. IEEE Trans. Consum. Electron. **56**(4), 2652–2660 (2010)
13. Bahari, A., Arslan, T., Erdogan, A.T.: Low-power H.264 video compression architectures for mobile communication. IEEE Trans. Circuits Syst. Video Technol. **19**(9), 1251–1261 (2009)
14. Lee, S., Kim, J.M., Chae, S.I.: New motion estimation algorithm using adaptively quantized low bit-resolution image and its VLSI architecture for MPEG2 video encoding. IEEE Trans. Circuits Syst. Video Technol. **8**(6), 734–744 (1998)
15. Chen, C.Y., Chien, S.Y., Huang, Y.W., Chen, T.C., Wang, T.C., Chen, L.G.: Analysis and architecture design of variable block-size motion estimation for H.264/AVC. IEEE Trans. Circuits Syst. I: Regul. Pap. **53**(3), 578–593 (2006)
16. Chen, T.C., Chen, Y.H., Tsai, S.F., Chien, S.Y., Chen, L.G.: Fast algorithm and architecture design of low-power integer motion estimation for H.264/AVC. IEEE Trans. Circuits Syst. Video Technol. **17**(5), 568–577 (2007)
17. Huang, Y.W., Chien, S.Y., Hsieh, B.Y., Chen, L.G.: Global elimination algorithm and architecture for fast block matching motion estimation. IEEE Trans. Circuits Syst. Video Technol. **14**(6), 898–907 (2004)

Chapter 7
Introduction to Scalable Image and Video Coding

The main aim of this chapter is to provide the fundamentals of wavelet based Scalable Video Coding (SVC), and to briefly discuss about its two widely followed variants, viz. Spatial Domain Motion Compensated Temporal Filtering (SD-MCTF) and In-Band Motion Compensated Temporal Filtering (IB-MCTF). This chapter starts with an overview of SVC, followed by discussion on the principle of MCTF. Next, Sect. 7.3 provides the details of the proposed framework for SVC. Simulation results are given in Sect. 7.4 and conclusions are drawn in Sect. 7.5.

7.1 Overview of Wavelet Based Scalable Video Coding

Scalability has drawn considerable attention of the researchers due to its capability of reconstructing low resolution or low quality video signal from partial bitstream. This enables a simple solution in adaptation to network and terminal capability. The partial bitstream is derived by dropping packets from the larger bitstream. A partial (subset) bitstream can represent a lower spatial resolution, or a lower temporal resolution, or a lower quality video signal (each separately or in combination) compared to the bitstream it is derived from.

Modern video transmission and storage systems using the Internet and mobile networks are typically based on Real-time Transport Protocol (RTP)/Internet Protocol (IP) for real-time services (conversational and streaming) and on computer file formats like MPEG-4 or 3GPP. Most RTP/IP access networks are typically characterized by a wide range of connection qualities and receiving devices. Such varying connection quality results from adaptive resource sharing mechanisms of these networks addressing the time varying data throughput requirements of a varying number of users. The variety of devices with different capabilities ranging from cell phones with small screens and restricted processing power to high-end PCs with high-definition displays results from the continuous evolution of these endpoints. Scalable video coding (SVC) is one solution to the problems posed by the characteristics

I. Chakrabarti et al., *Motion Estimation for Video Coding*,
Studies in Computational Intelligence 590, DOI 10.1007/978-3-319-14376-7_7

of modern video transmission systems. The applications like streaming, conferencing, surveillance, broadcast and storage can benefit from SVC.

7.1.1 Existing Scalable Video Codec Designs

Research on scalability is carried out in one or a combination of more than one of the following domains, viz. temporal, spatial, and quality. In particular, spatial and temporal scalability are described as a subset of the bit stream that represents the source content with a reduced picture size (spatial resolution) or frame rate (temporal resolution), respectively. With quality scalability, the substream provides the same spatio-temporal resolution as a complete bit stream, though with a lower signal-to-noise ratio (SNR).

Scalable video coding must support more than one spatial, temporal and quality layers. Hence, the codec structure of SVC differs from the conventional hybrid video coding structure of the H.264 video standard. There have been many contributors to the codec structure for an SVC. The first significant contribution to the SVC codec structure was made by Ohm [1]. The following video codec structures (Figs. 7.1, 7.2 and 7.3) are examples of video codec structures developed by Ohm [1].

Spatial Scalability: Video is coded at multiple spatial resolutions. The data and decoded samples of lower resolutions can be used to predict data or samples of higher resolutions in order to reduce the bit rate to code the higher resolutions. Spatial scalability is typically realized as a differential pyramid, where motion compensated prediction is applied within each pyramid level in addition to the coarse-to-fine prediction. There may be cases, however, where the reconstruction of the previous frame enhancement layer allows better prediction of the actual enhancement frame, without referencing the current base-layer frame. As shown in Fig. 7.1, the enhancement layer frame can either be predicted entirely from the up-sampled base layer, from the previous enhancement layer reconstruction, or from the mean value of both.

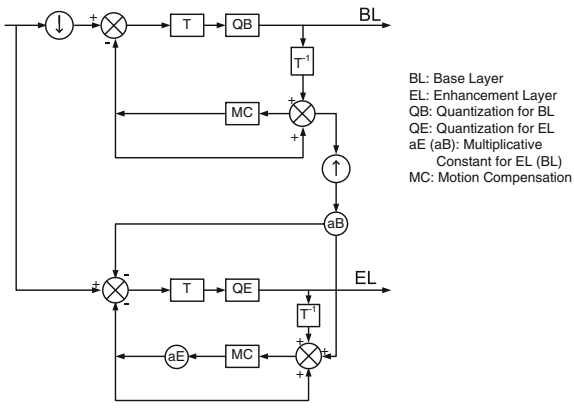


Fig. 7.1 Block diagram of spatial scalable codec (*Source* Ohm [1])

Temporal Scalability: It is often used in practice, as reduction of the video frame rate is a common approach in cases where insufficient transmission capacity is available. Assume that the base layer relates to a reconstructed sequence of lower frame rate. If the base information is self-contained, it can be established as a subsequence from which frames are skipped, while the enhancement layer supplements these frames for the higher frame rate, which are then predicted from the base-layer frames as depicted in Fig. 7.2.

SNR/Quality/Fidelity Scalability: Video is coded at a single spatial resolution though at different qualities. The data and decoded samples of (base layer) lower qualities can be used to predict data or samples of the (enhancement layer) higher qualities in order to reduce the bit rate to code the higher qualities. To achieve higher compression performance, interframe prediction with a separate loop can be applied to the enhancement layer coding as shown in Fig. 7.3.

The above three block diagrams (Figs. 7.1, 7.2 and 7.3) were a diagrammatic representation of the video codec proposed by Ohm [1]. All these video codec designs were modifications of the conventional hybrid coding [2–7]. Hybrid coding has been prevalent in all the video coding standards since the introduction of motion compensation for video coding. In case of encoding of the hybrid video structure, one frame predicts another, the predicted frame predicts another and this goes on for a Group-of-Pictures (GOP). It is quite evident and stated in [7] that prediction error tends to accumulate and the quality of the frames worsens as we move towards the last frame of the GOP. To avoid such a problem, Motion Compensated Temporal Filtering has been introduced [8–11].

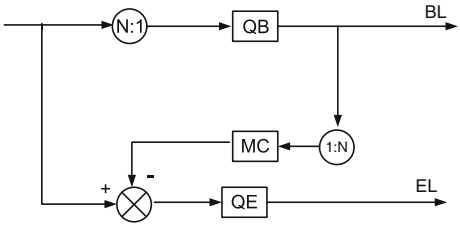


Fig. 7.2 Block diagram of temporal scalable codec (*Source* Ohm [1])

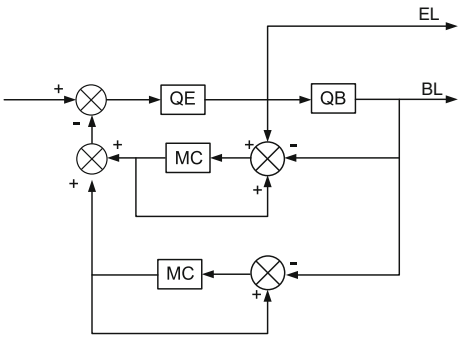


Fig. 7.3 Block diagram of SNR scalable codec (*Source* Ohm [1])

Application of motion compensation (MC) is a key for high compression performance in video coding. It is often understood to be implicitly coupled with frame prediction schemes. There is indeed no justification for this restriction, as MC can rather be interpreted as a method to align a filtering operation along the temporal axis with a motion trajectory. In the case of MC prediction, the filters are in principle linear predictive coding (LPC) analysis and synthesis filters, while in cases of transform or wavelet coding, the transform basis functions extended over the temporal axis are subject to MC alignment. This is known as motion-compensated temporal filtering. If MCTF is used in combination with a 2-D spatial wavelet transform, this shall be denoted as a 3-D or (depending on the sequence of the spatial and temporal processing) either as a 2-D + t or t + 2-D wavelet transform. In case of MCTF (Fig. 7.4), the error frame is used to update the reference frame. Hence, the error remains within the candidate and reference frames and is not accumulated or propagated to the successive frames.

Andreopoulos et al. [8, 9], introduced wavelet-based Scalable Video Coding. Instead of the conventional discrete cosine transform (DCT), discrete wavelet transform (DWT) was introduced as a suitable Image and Video Transform. DWT is a multi-resolution transform, and by using this property it provides an improved representation of the digital video data in a hierarchical manner. This is a useful property which can be extensively used in case of SVC. The first codec structure which introduced the property of MCTF in SVC was SD-MCTF [8]. SD-MCTF performs motion compensation in temporal axis of the pixel domain. The residual frames were then spatially decomposed using a suitable Discrete Wavelet Filter. For better coding efficiency [1], IB-MCTF [8, 9] was introduced. As motion compensation was performed in the wavelet domain, the problem of shift-variance can not be avoided. In order to solve this problem, Over-complete DWT [8, 9, 12, 13] has to be performed. The algorithm and complexity models of IB-MCTF were given in [9, 12]. Conventional wavelet decomposition was performed using the convolution method. However, the problem with convolution based DWT was higher memory requirement and greater computational time. To solve this problem, a mathematical approach called lifting scheme was introduced by Daubechies and Sweldens [14] through factorization

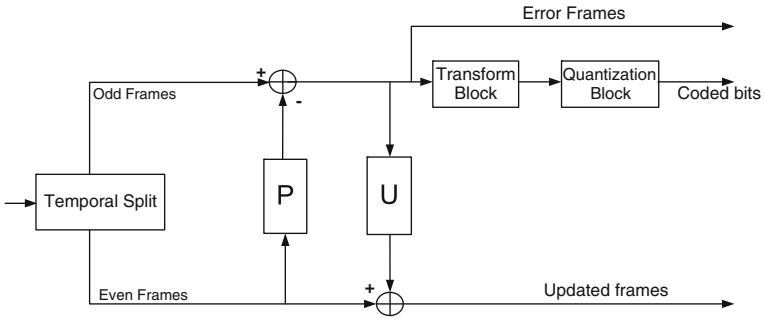


Fig. 7.4 Motion compensated temporal filtering [8, 9] block with predict and update stage

of Discrete Wavelet Filters. Details of over complete DWT (ODWT), lifting based ODWT, SD-MCTF, and IB-MCTF are given in the following sections.

7.1.2 Discrete Wavelet Transform

Fourier Transform of any signal generates the frequency profile of the signal. It shows the frequency components that are present in the signal. It fails to show spatial or temporal variation of the frequency component, i.e. it cannot show the position of the frequency at the time or space axes. Wavelet Transform [15, 16] on the other hand decimates the signal both in frequency as well as in time/spatial domain; it captures both the frequency and location information. This is a key advantage of wavelet transform over Fourier transform. DWT is the representation of signals as a series summation of certain wavelets. Some popular wavelets that are used in the context of DWT are Haar wavelets, Daubechies wavelets, 5/3 orthonormal wavelets and 9/7 orthonormal wavelets. The feature that makes DWT suitable in the context of SVC is its ability to decompose the signal into lower resolution. DWT is also called Multi-resolution Transform. The signal passes through two filters—a high pass filter that extracts the high frequency components (detailed coefficients) and a low pass filter that produces the low frequency components (approximate coefficients). The approximate coefficients are the representation of the signal in the lower resolution in the spatial domain. The low frequency components can be further decomposed to give even lower resolution in the spatial domain. This property of DWT to decompose a signal into lower resolution makes it suitable in the context of SVC. However, wavelet transform suffers from a problem of shift variance which is the reason of the dominance of DCT in the video coding standards. This problem of shift variance is the reason why motion estimation [17] algorithm cannot be directly used in the wavelet domain representation of consecutive frames.

7.1.3 Problem of Shift Variance in DWT

The representation of any signal using the DWT gives both frequency and location information. This property is desirable in case of image and video coding applications. The block diagram of the wavelet decomposition has a down-sample stage. Certain samples from both the approximate coefficients (low pass) as well as the detailed coefficients (high pass) are dropped leading to lower resolution. This shows that wavelet decomposition is highly dependent on the alignment of the signal and the discrete grid chosen for analysis. It is clear that a unit shift in the spatial domain does not correspond to a unit shift in the wavelet domain. This can cause a lot of high frequency components to develop at the edges of the wavelet domain signal while performing motion compensation in the wavelet domain. This problem is called the shift variance.

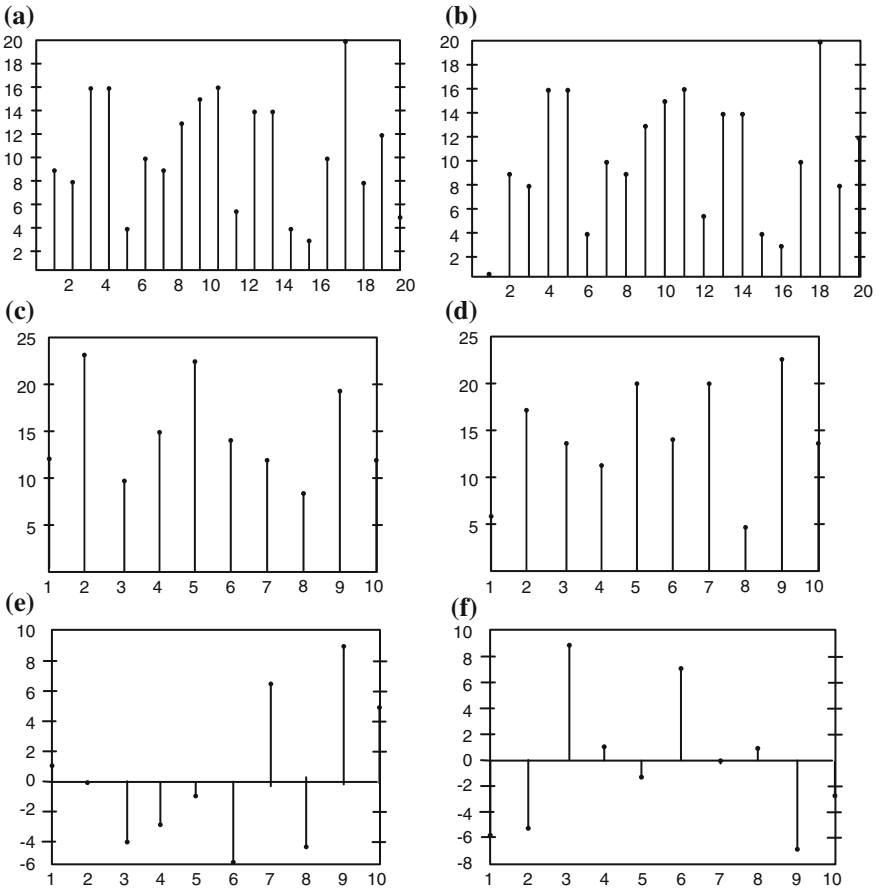


Fig. 7.5 Problem of shift variance in DWT. Parts **a** and **b** show a one dimensional signal $x[n]$ and its time shifted version. Parts **c** and **d** show low pass coefficients of $x[n]$ and $x[n-1]$. Parts **e** and **f** depict high pass coefficients of $x[n]$ and $x[n-1]$

Figure 7.5 shows that the wavelet domain representation of the detailed and approximate coefficients is quite different for any signal and its unit time shifted version. Let $y[n]$ be equal to DWT($x[n]$), where $x[n]$ is a one dimensional signal. Due to shift variance, $y[n-1]$ is not equal to DWT($x[n]$). This difference is more prominent in case of the detailed coefficients than the approximate coefficients. At the edges of any signal in the spatial domain, this disparity is most critical and causes a lot of high frequency components to develop at those positions. So if there is a unit time shift in any signal, it cannot be captured by the critically sampled Discrete Wavelet Transform. A single pixel shift in either direction or both cannot be captured by the critically sampled wavelet transform. So for effective motion compensation in the

wavelet domain, we need to obtain the phase information as well. This can be done by performing the operation of Over-complete Discrete Wavelet Transform (ODWT).

7.1.4 Critically Sampled DWT

The DWT of a signal $x[n]$ is calculated by passing it through a series of filters. First, the samples are passed through a low pass filter with impulse response $g[n]$ resulting in a convolution of the two:

$$y[n] = (x * g)[n] = \sum_{k=-\infty}^{\infty} x[k]g[n-k] \tag{7.1}$$

The signal is also decomposed simultaneously using a high-pass filter. The outputs represents the detail coefficients (from the high-pass filter) and the approximation coefficients (from the low-pass). It is important that the two filters are related to each other and they are known as quadrature mirror filter [15, 16]. However, since half the frequencies of the signal have now been removed, half the samples can be discarded according to Nyquist criteria. The filter outputs are then down sampled by two.

$$\begin{aligned} y_{low}[n] &= \sum_{k=-\infty}^{\infty} x[k]g[2n-k] \\ y_{high}[n] &= \sum_{k=-\infty}^{\infty} x[k]h[2n-k] \end{aligned} \tag{7.2}$$

This decomposition has halved the time resolution since only half of the output of each filter characterizes the signal. However, as each output has half the frequency band of the input, the frequency resolution has been doubled. Because of the down-sampling operation, this method of DWT is called critically sampled DWT or complete DWT.

7.1.5 Over-Complete Discrete Wavelet Transform (ODWT)

The transformation based on ODWT has been used to overcome the problem of shift variance faced by critically sampled DWT. In the critically sampled DWT structure, in the down-sample stage, odd terms of the signals are discarded and the even terms of signals are retained. This method is known as even phase DWT. If we retain the odd terms and discard the even terms, then the DWT is odd phase DWT. In over-complete DWT, both the even phase and the odd phase are retained. This implies that there is no down-sample stage since it is the interlaced version of even phase DWT and odd phase DWT.

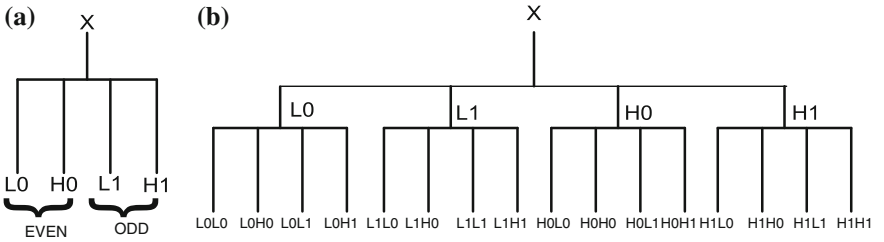


Fig. 7.6 Block diagram of over-complete discrete wavelet transform **a** 1-D signal. **b** 2-D signal

Figure 7.6a shows that if X is an one dimensional (1-D) signal, after the first level of wavelet decomposition using ODWT, we obtain four sub-bands instead of two phases (even and odd). However, if the signal is a two dimensional (2-D) signal, like an image, this over-complete DWT has been performed at each decomposition stage, the row-wise decomposition stage and the column-wise decomposition stage. As shown in Fig. 7.6b various sub-bands in the case of an image would be LL, LH, HL and HH sub-bands. The number of phases increases as one goes down the resolution levels. For a 1-D signal, the number of phases at a certain decomposition level I is $2I$, i.e. 2 phases in the first level and four phases in the second level and so on. Similarly, for a 2-D signal (image), the number of phases at a certain level is 2^{2I}, i.e. four phases in the first level, sixteen phases in the second level, and so on.

7.1.6 Lifting Based Discrete Wavelet Transform

Implementation of convolution based DWT requires very intense computation and a lot of memory. These features of DWT make it unfavorable in the field of video compression and video coding. To avoid the problems of convolution based DWT, a mathematical approach called the lifting-based wavelet transform or simply lifting has been developed. The main feature of this method is to factorize the low pass and the high pass filters into a number of mathematical operations. The method uses spatial correlation between the high pass and the low pass filter coefficients to minimize computations and improve the system performance. The operations involved are sequential, as memory requirement is quite nominal as there is no data dependency between the input and the output. Hence, the value of the transformed data can be loaded back into the same memory. Figure 7.7 shows the block diagram of a lifting scheme. Haar wavelets have only one Predict Stage (P) and one Update stage (U). However, other wavelets like D4, 5/3 orthonormal and 9/7 orthonormal wavelets have more than one Predict and Update stages. Irrespective of the number of Predict and Update stages, every lifting scheme has a similar framework. The first step of any lifting scheme is a split. In splitting stage the input sequence To perform the lifting scheme, the first step is splitting the input bit-stream $x[n]$ separated into a $x[2n]$

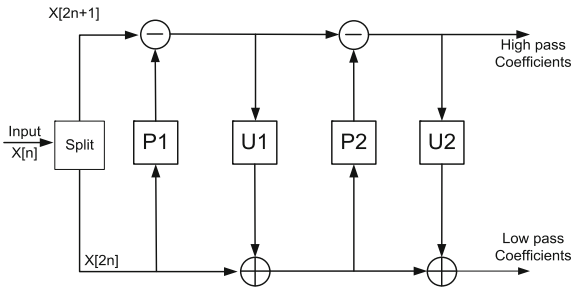


Fig. 7.7 Block diagram of lifting based DWT for an one dimensional signal

(even) and $x[2n+1]$ (odd) coefficients. Always even signal coefficients predict the odd signal coefficients. The difference between the odd coefficients and the predicted coefficients is used to update the even coefficients. So along the odd coefficient data path, the high pass coefficients are obtained. Along the even coefficient data path, the low pass coefficients are obtained. For a two dimensional (Image) input sequence, $x[n]$ split in to even and odd coefficients and then the set of predict and update stages is performed to obtain the low Pass and the high pass coefficients along the row. Next, both the high pass and low pass coefficients are passed through a similar structure as depicted in Fig. 7.8, this time the frames are split along the column. So after the predict and update stages are applied along the column, four sub-bands are obtained, viz. LL, LH, HL and HH.

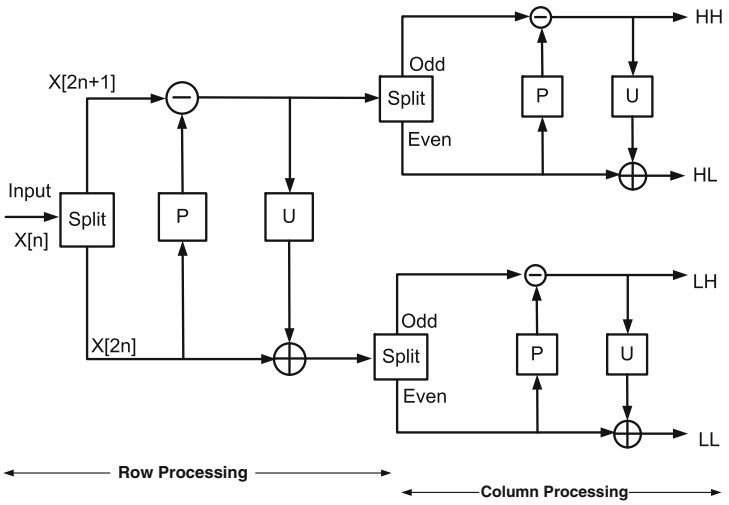


Fig. 7.8 Block diagram of lifting based DWT for a 2-D signal (Image)

7.1.7 Over-Complete Discrete Wavelet Transform Using the Lifting Scheme

The importance of over complete DWT in the perspective of Motion compensation (MC) in the wavelet domain already has been mentioned earlier. Lifting based over complete DWT is a method [12] proposed to solve the problem of shift variance in DWT using the lifting scheme.

As shown in Fig. 7.8, the first stage splits the frame into odd and even coefficients. Then the even coefficients are used to predict the odd coefficients and the difference of that data is used to update the even coefficients. Note that the even coefficients are members of the set $\{x_0, x_2, x_4, \ldots, x_{2n}\}$, while the odd coefficients are $\{x_1, x_3, x_5, \ldots, x_{2n+1}\}$. The wavelet coefficients are a function of the data-sets $(x_0, x_1), (x_2, x_3), (x_4, x_5), \ldots, (x_{2n}, x_{2n+1})$. However, no information is available for the dataset $(x_1, x_2), (x_3, x_4), (x_5, x_6), \ldots, (x_{2n-1}, x_{2n})$. For normal lifting based DWT, the even coefficients predict the odd coefficients and the difference data is used to update the even coefficients. For the lifting based over complete DWT, same process done for one pixel shifted version of the signal. These two processes when performed together gives the over-complete DWT coefficients. For the scenario of an image, the lifting framework remains the same but for the fact that the operation is performed first along the row and then along the column as well.

7.1.8 Spatial Scalability with DWT

Apart from the decimation in space and frequency, the other advantage of wavelet transform over any other conventional transform is its ability to decompose the signal into multiple resolutions. In the previous section, the architectures for lifting based DWT and lifting based ODWT have been mentioned. In this section, we briefly discuss how DWT can be used for spatial scalability. After the first level of decomposition, we obtain four sub-bands—LL, LH, HL and HH. The LL sub-band is the representation of the image at a lower spatial resolution, which is of size equal to $1/2 \times 1/2$ of the original image. The other sub-bands contain the high pass information or the detailed information. We can use the LL to decompose the image further. After the second level of decomposition, we have LLLL sub-band which is of size equal to $1/4 \times 1/4$ resolution of the original image. We can decompose further, giving an even lower resolution image of resolution $1/8 \times 1/8$ of that of the original image. This sub-band will be the base-layer for the spatial scalable video bit-stream. The other sub-bands become the enhancement layers. The details are depicted in the Fig. 7.9.

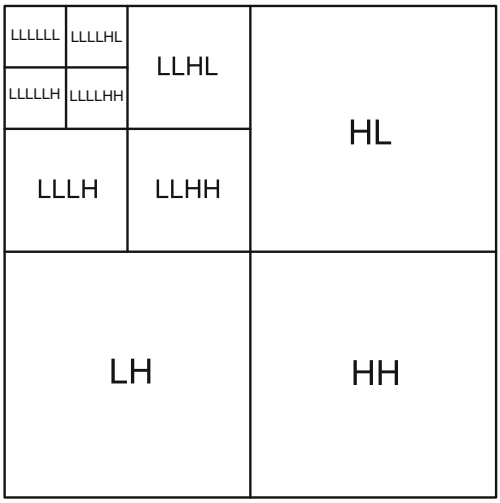


Fig. 7.9 Diagrammatic Representation of spatial decomposition using DWT showing four different resolution levels

7.1.9 Temporal Scalability with DWT

Utilizing the multi-resolution property of wavelet transform along the temporal axis helps obtain temporal scalability. For temporal transform, the inter-frame wavelet coding has been proposed earlier which however fails to meet the compression demands of today's broadcasting network. But keeping the basic structure of inter-frame wavelet coding, MCTF has been proposed which has been seen [11, 12] to give better coding efficiency. After one level of temporal decomposition of each pair of frames, we obtain one low pass or average frame and one high pass or difference frame. This low pass frames can be combined together to form the second level of temporal decomposition. The high pass frames are kept separate. So after two levels decomposition, we have one low pass frame and three high pass frames. The low pass frames are the average over four consecutive frames. By combining these two low pass frames, we may decompose the video signal further. At the end of three levels of decomposition, we have one low pass frame and seven high pass frames. This low pass frame or LLL frame becomes the base layer for temporal scalability. As shown in Fig. 7.10, for every eight frames, we have one such LLL frame which is the average over these eight frames. This way the frame rate of this LLL frame is 1/8th of the original frame rate. The other high pass frames becomes the enhancement layers. This is the procedure for temporal scalability.

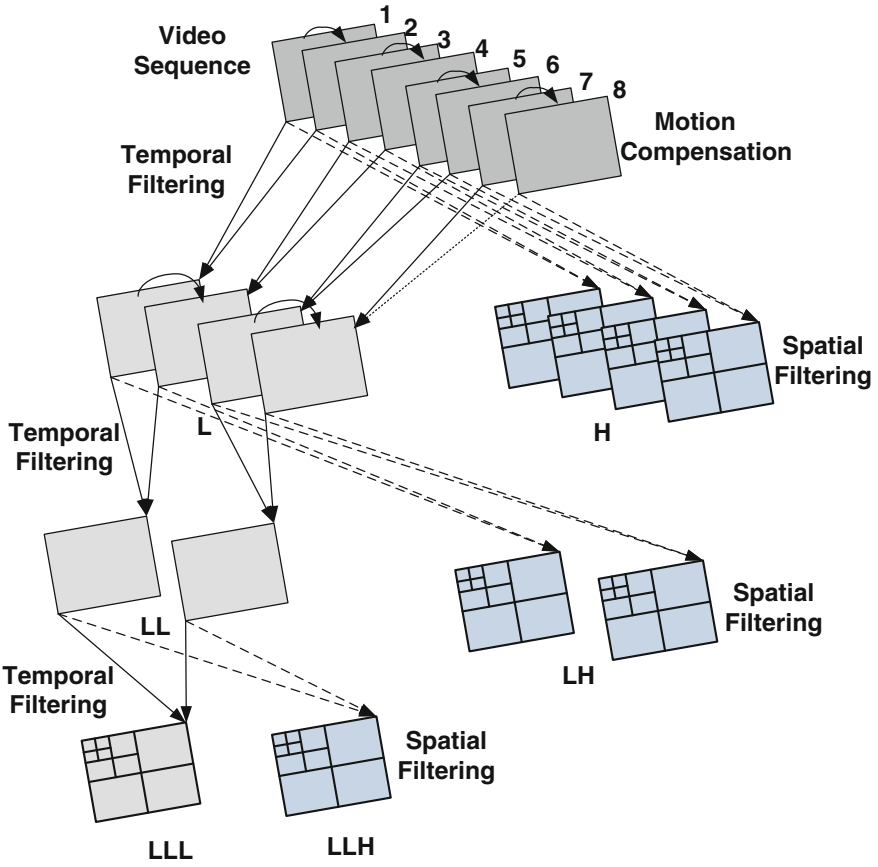


Fig. 7.10 Wavelet-based motion-compensated temporal filtering

7.2 Motion Compensated Temporal Filtering (MCTF)

Hybrid video coding, as found in H.264/AVC [6, 7] and all existing video coding designs are based on differential pulse code modulation (DPCM) together with spatial decorrelating transformations [7]. DPCM is characterized by the use of synchronous prediction loops at the encoder and decoder. Differences between these prediction loops lead to a "drift" that can accumulate over time and produce annoying artifacts. Only recently, new methods of efficient enhancement layer prediction represented by the motion-compensated wavelet coding were developed to improve the traditional hybrid scalable coders. These new wavelet codecs have been considered a useful coding structure with numerous advantages over non-scalable conventional techniques based on motion-compensated prediction [18–20].

One key element of recent advances in scalable video coding is the hierarchical prediction structure characterized MCTF which involves lifting based DWT. In lifting-based MCTF framework, a wavelet transform is applied along the motion trajectories and the recursive prediction loop in conventional hybrid video coding is replaced with an open loop structure [1]. The wavelet transform is efficiently implemented by lifting scheme, which is easily invertible and any type of operation, linear or non-linear, can be incorporated into the prediction and update steps. As a consequence, these wavelet video coding schemes are believed to provide flexible spatial, temporal, SNR and complexity scalability with fine granularity over a wide range of bit-rates, while maintaining a very high coding efficiency. In the future, the design of the scalable extension of the HEVC may include a wavelet decomposition structure in the temporal direction, because of its high quality.

7.2.1 Spatial Domain MCTF (SD-MCTF)

SD-MCTF is the first motion compensated temporal filtering algorithm that has been proposed. This has been the first ever modification of the conventional hybrid coding. Inspite of its high efficiency hybrid coding fails in error resilience. Due to the loop structure, the error tends to accumulate within the Group-of-Pictures (GOP). To avoid this, a non-loop/open loop structure has been proposed in [9]. In SD-MCTF, The motion compensation is performed in the spatial domain. Two consecutive frames, the odd numbered frame being the candidate frame and the even numbered frame being the reference frame are considered. Using a suitable block matching algorithm [17], the predicted frame is obtained. The better the prediction, the better will be the compression. This is because the number of significant bits in the error frame will be less. Some pixels in the reference frame get mapped to more than one pixel in the candidate frame, whereas the other pixels are not mapped to any one of the pixels in the candidate frame. The error frame is used to update the reference frame. For the update process, the information of the inverse motion vector is necessary. This information can be calculated from the motion vector information. The updated frame is obtained using the same algorithm as is used for the predicted frame. The algorithm of SD-MCTF comprises several steps as follows:

Step 1 The sequence of frames is split into even and odd frames. Even frames become the reference frames and the odd frames become the candidate frames. Motion estimation using a suitable block matching algorithm is applied. The output of the ME block is the motion vector information.

Step 2 Using the information from the motion vector, a predicted frame is obtained. The best matched macro-block from the reference frame is copied into the position of the candidate MB in the predicted frame.

Step 3 The next step is to obtain the error frame. It is a pixel-by-pixel subtraction of the predicted frame from the candidate frame. The sum-squared

pixel-by-pixel value of the error frame gives the measure of the accuracy of ME/MC algorithm.

Step 4 The next step is to obtain the updated frame. This is the part that has been introduced in MCTF. Until step 3, the procedure is same for both the hybrid structure as well as the MCTF. Update step involves creating an update frame using the information from the error frame and inverse motion vector. The inverse motion vector can be obtained from forward motion vector. Obtaining an update frame is similar to step 2. While calculating the inverse motion vector, it can be observed that some of the Macro-Blocks in the reference frame are multi-connected while some have no connection, or there exists no reference in the motion vector. For the multi-connected MBs in the reference frame, the inverse motion vector obtained first using the raster scan is applied. For the non connected MBs in the update frame for those pixels, the value of motion vector is zero.

Step 5 The next step is performing the averaging. The pixel values of the update frame are divided by 2, and this is added to the pixel values of the corresponding position of the reference frame. After this step, we obtain the average frame.

Step 6 Repeat steps 1–5 with video stream inputs as the average frames or the low pass frames obtained from the previous decomposition to further decompose the video stream into even lower temporal resolutions.

Step 7 Following the decomposition of the even and odd frames into average and difference frames respectively, spatial decomposition is performed to remove the spatial redundancy. DWT is used for spatial decomposition. Any one of the wavelets (like Haar, D4, and different orthonormal wavelets) can be used as a suitable basis for DWT.

Spatial Domain MCTF is also called 't + 2D' transform. Temporal decomposition is carried out first, followed by a spatial transform.

7.2.2 In-Band MCTF (IB-MCTF)

Figure 7.11 represents the block diagram for IB-MCTF. In this scheme, the motion estimation [13] is performed in-band or in the sub-bands. In case of IB-MCTF, the motion estimation is performed in-band, and the performance of the transcoder suffers from the problem of shift variance. Because DWT is shift variant, it is not possible to use the critically sampled DWT sub-bands for motion estimation. So for effective motion compensation in the wavelet domain, we need to use over-complete DWT. Apart from the sequence of temporal and spatial decomposition and performing ODWT instead of the critically sampled DWT, the basic framework of IB-MCTF [9] is similar to that of SD-MCTF. IB-MCTF also has the same structure for MCTF except for Motion Vector (MV) information and Inverse Motion Vector (IMV) information, in which the phase information has to be computed. IB-MCTF is also called 2D + t transform.

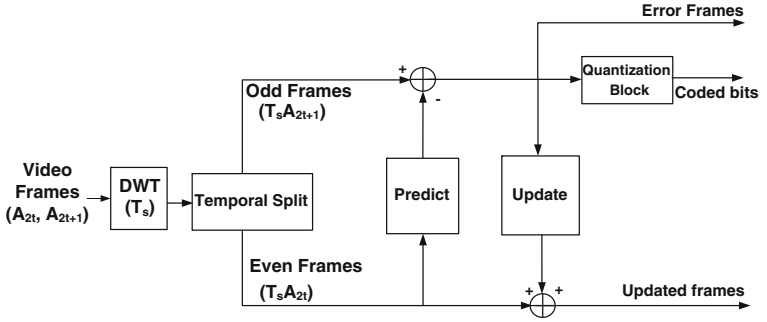


Fig. 7.11 Block diagram for IB-MCTF with predicting and update blocks

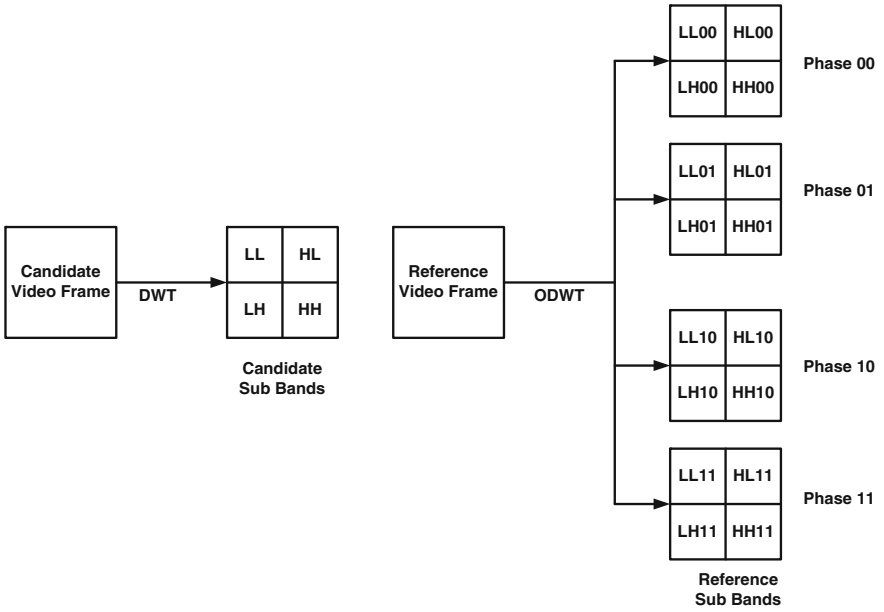


Fig. 7.12 Step 1: Spatial decomposition to obtain the sub-bands; critically sampled DWT for candidate frame and ODWT for reference video frame

The algorithm of IB-MCTF consists of a number of steps as follows:

Step 1 The first step of IB-MCTF involves splitting of the video frame into the sub-bands. Because of the problem of shift variance, ODWT is performed to obtain the phase information. The even frames are the reference frames (Fig. 7.12).

Step 2 Motion estimation is performed on the sub-band LL of the candidate frame and the reference frame. For motion estimation, the information that is sent to the ME block is the candidate sub-band, and the reference sub-bands for

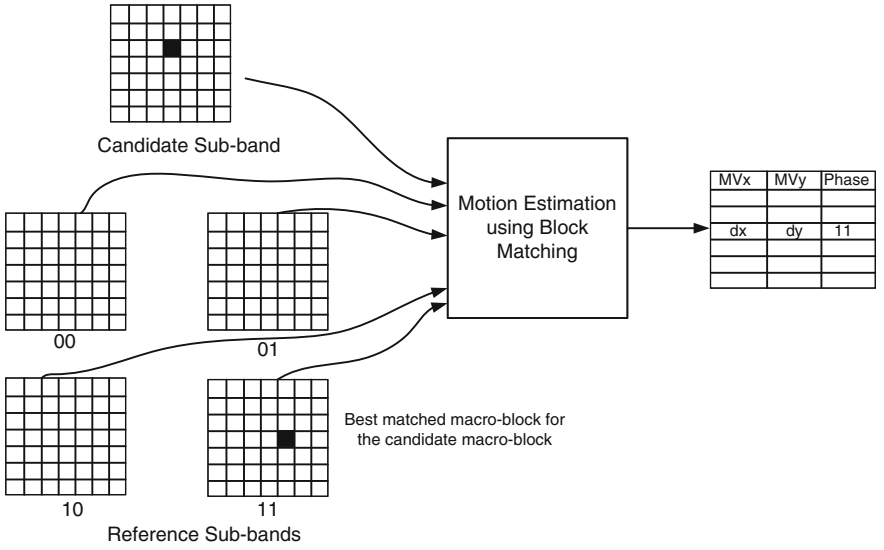


Fig. 7.13 Step 2: Motion estimation process in case of IB-MCTF, the phase information also considered for ME

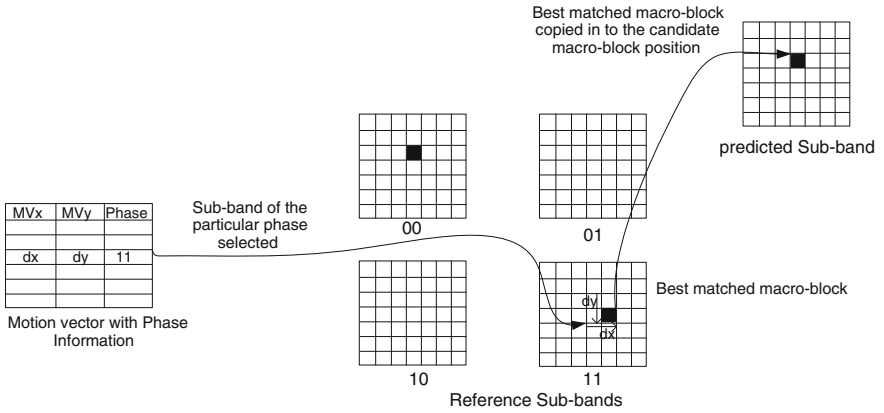


Fig. 7.14 Step 3: Predicted sub-band obtained using information from the reference sub-band and the motion vector data along with false information

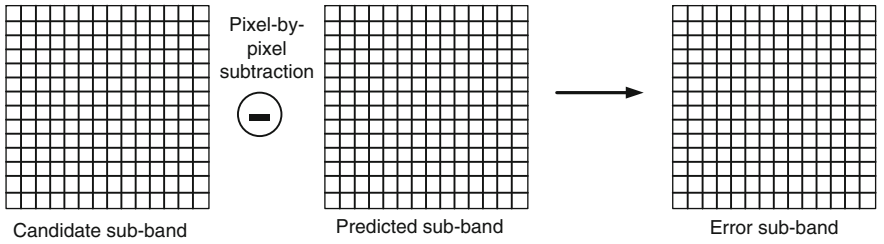


Fig. 7.15 Step 4: Error sub-band by subtracting predicted sub-band from the candidate sub-band

all the phases (00, 01, 10 and 11). ME is performed using the block matching algorithm, and the output of the ME block is the motion vector. In addition to the motion vector, phase information is obtained (Fig. 7.13).

Step 3 Using the phase information and the motion vector, a predicted sub-band is obtained. The best matched macro-block from the reference sub-band of the particular phase is copied into the position of the candidate MB in the predicted sub-band (Fig. 7.14).

Step 4 The error sub-band is a pixel-by-pixel subtraction of the predicted sub-band from candidate sub band (Fig. 7.15).

Step 5 The next step is the formation of the update sub-band. For this process, the inverse motion vector is obtained. The process for IMV is not as straightforward as in case of SD-MCTF. The algorithm to obtain the inverse motion vector has been mentioned in [9, 12]. If the phase of the best match is 00, then the inverse motion vector is obtained by the same procedure as in SD-MCTF. However, for non-zero phase, the inverse motion vector is modified depending on the phase information. The problem of multi-connected and non-connected macro-blocks is still present in the case of IB-MCTF. Unlike the predict stage where the different phases of the reference sub-band are present, the different phases of the error sub-band are unavailable at this stage. Here, CODWT is performed. The phase information and the inverse motion vector are used to form the updated sub-band (Figs. 7.16 and 7.18).

Step 6 This is the averaging step. In this step, the reference sub-band is added to the updated sub-band/2. Each pixel value in the average sub-band is formed by

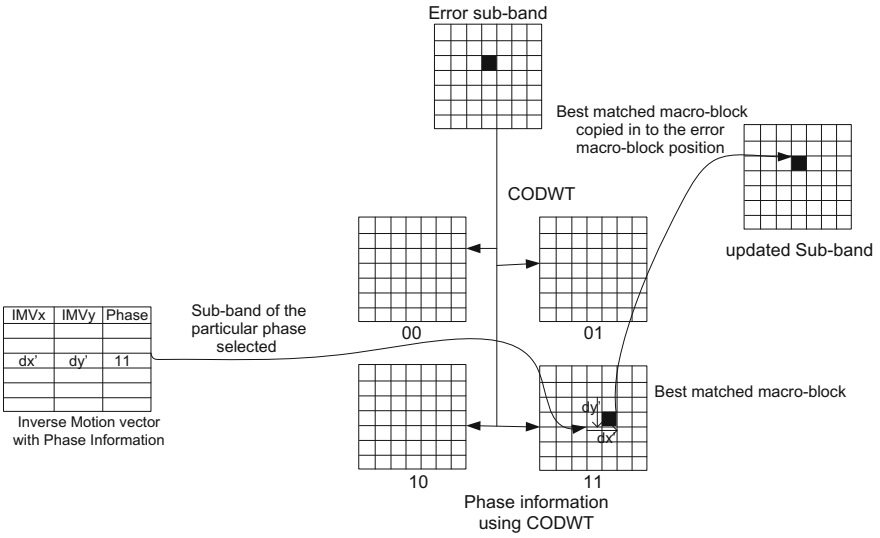


Fig. 7.16 Step 5: Formation of the update frame from the error frame using IMV and phase

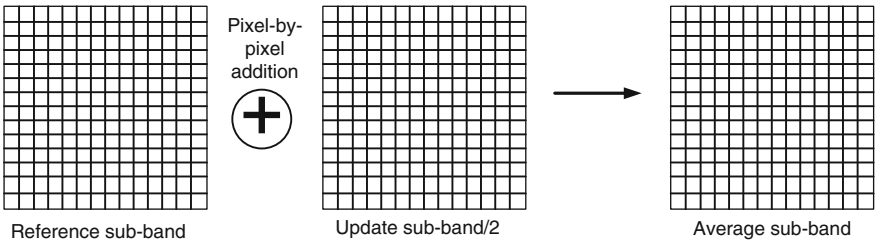


Fig. 7.17 Step 6: Formation of the average sub-band by adding the reference sub-band to the update sub-band/2

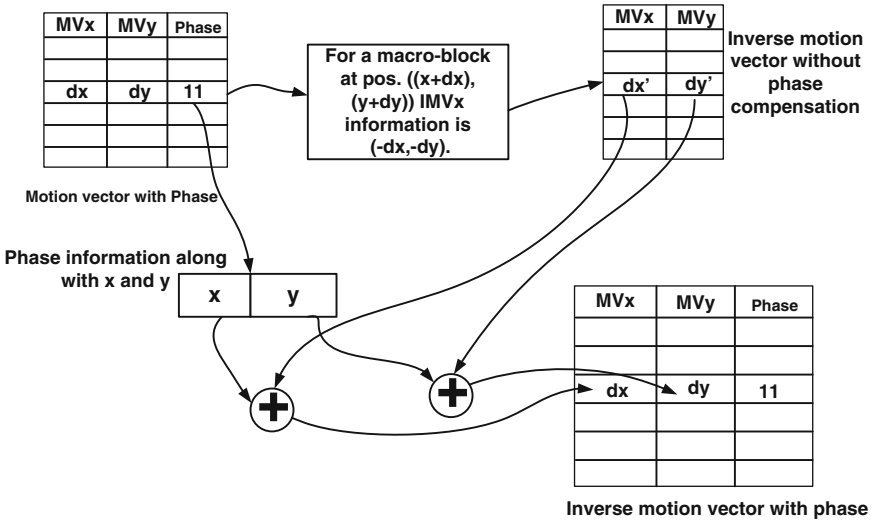


Fig. 7.18 Inverse motion vector with phase information from motion vector using IB-MCTF algorithm mentioned in [9]

the sum of the pixel value of reference sub-band and 1/2 value of the pixel of the updated sub-band (Fig. 7.17).

Step 7 The previous steps of IB-MCTF algorithm are performed for decomposing the video stream for one spatial and one temporal level. For further decomposition, step 1 should be repeated for further spatial decomposition levels and then Step 2 to Step 6 should be repeated for subsequent temporal levels.

7.3 Proposed Framework for SVC

The main encoding framework for scalable video coding based on IB-MCTF is illustrated in Fig. 7.19. Initially, we apply lifting based DWT on each frame of the GOP. The size of the GOP is 8. The number of levels in DWT depends on the

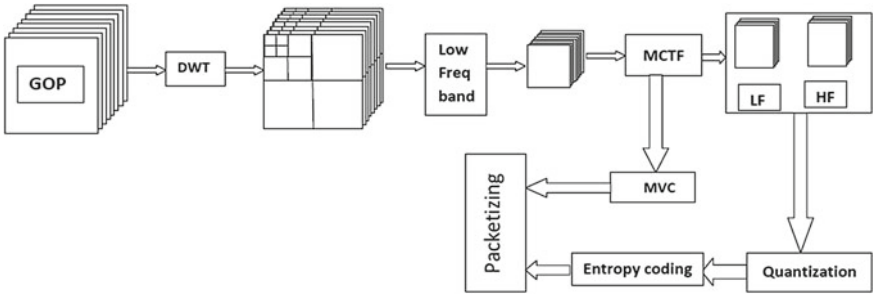


Fig. 7.19 Proposed framework for SVC based on IB-MCTF

resolution of the base layer. In our framework, we consider one level decomposition using CDF 9/7 filter coefficients to perform the DWT operation. MCTF is applied in the temporal direction on low frequency (LF) bands of each pair of frames in a GOP. In this process, each odd frame is predicted from the even frame through MC. Because MC is applied on DWT coefficients of the image, we have adopted low band shift method for motion estimation [12]. To avoid the problem of shift variance in DWT, we have used the ODWT. After the prediction step, the even frame was updated by using CODWT [9]. After the first level of temporal decomposition, we get 4 LF frames and 4 HF frames. For the second level, 4 LF frames are the inputs, from which we get 2 LF and 2 HF frames. At the third level, we get 1 LF and 1 HF frame. The maximum number of levels in temporal direction is three, as the GOP size is eight. After the 3rd level decomposition in temporal direction, we have 1 LF and 7 HF frames. Entropy coding is next applied to those frames. We consider EZW+Huffman coding for entropy coding. Finally, coded motion vectors and output of entropy coder are subjected to packetization. The bi-orthogonal 9/7 wavelet can be implemented as four lifting steps followed by scaling (shown in Fig. 7.20). It entails the hardware implementation of six equations embodied by Eq. (7.3).

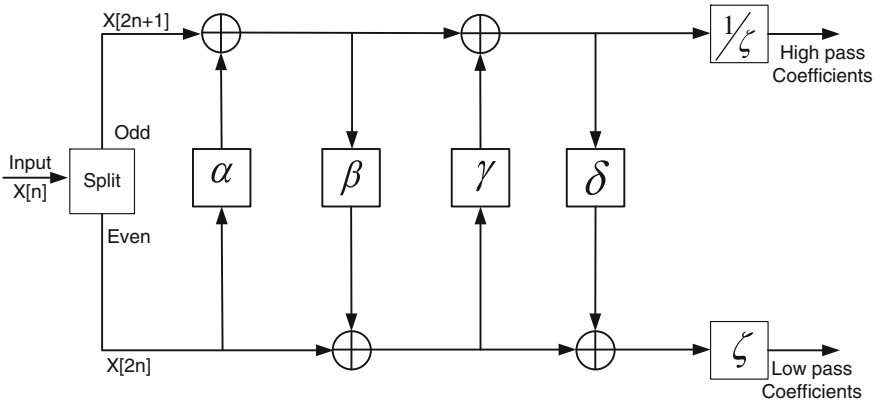


Fig. 7.20 1-D lifting scheme of Daubechies 9/7 for forward wavelet DWT

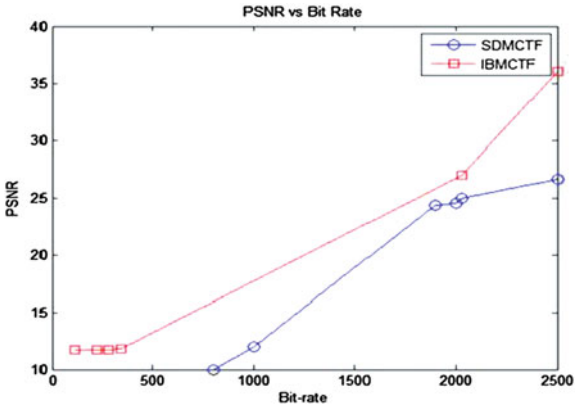


Fig. 7.21 PSNR (in dB) versus bit-rate (in kbps) for IB-MCTF and SD-MCTF

$$
\begin{aligned}
X_1[2n+1] &\leftarrow X[2n+1] + \alpha\{X[2n] + X[2n+2]\} \\
X_2[2n] &\leftarrow X[2n] + \beta\{X_1[2n+1] + X_1[2n-1]\} \\
X_3[2n+1] &\leftarrow X_1[2n+1] + \gamma\{X_2[2n] + X_2[2n+2]\} \\
X_4[2n] &\leftarrow X_2[2n] + \delta\{X_3[2n+1] + X_3[2n-1]\} \\
X_5[2n+1] &\leftarrow 1/\zeta\{X_3[2n+1]\} \\
X_6[2n] &\leftarrow \zeta\{X_4[2n]\}
\end{aligned} \tag{7.3}
$$

The original data to be filtered is denoted by $X[n]$; and the 1-D DWT outputs are the detail coefficients $X_5[n]$ and the approximation coefficients $X_6[n]$. The lifting step coefficients are denoted by α, β, γ and δ and the scaling coefficient is represented by ζ are constants [14]. The above equations are implemented on Matlab to obtain the coefficients $X_5[n]$ and $X_6[n]$, which correspond to high pass and low pass coefficients respectively. For an image, which is a 2-D signal, the above process is performed in rows and as well as columns.

7.4 Simulation Results

We have simulated scalable video codec architecture based on IB-MCTF and SD-MCTF through Matlab tool. For both the architectures, we consider the size of the GOP as eight, the number of decomposition levels in the spatial domain as one. We used lifting based CDF 9/7 filter coefficients for wavelet. For SVC based IB-MCTF, motion compensation and motion estimation are done on wavelet domain, whereas in SD-MCTF it was done in pixel domain. In temporal axis, always an odd frame is predicted by even frames through lifting based prediction step. Prediction step involves the ME and MC. After this, the even frames are updated by residual frames (HF frame). After the one level of temporal decomposition we get 4 LF frames and

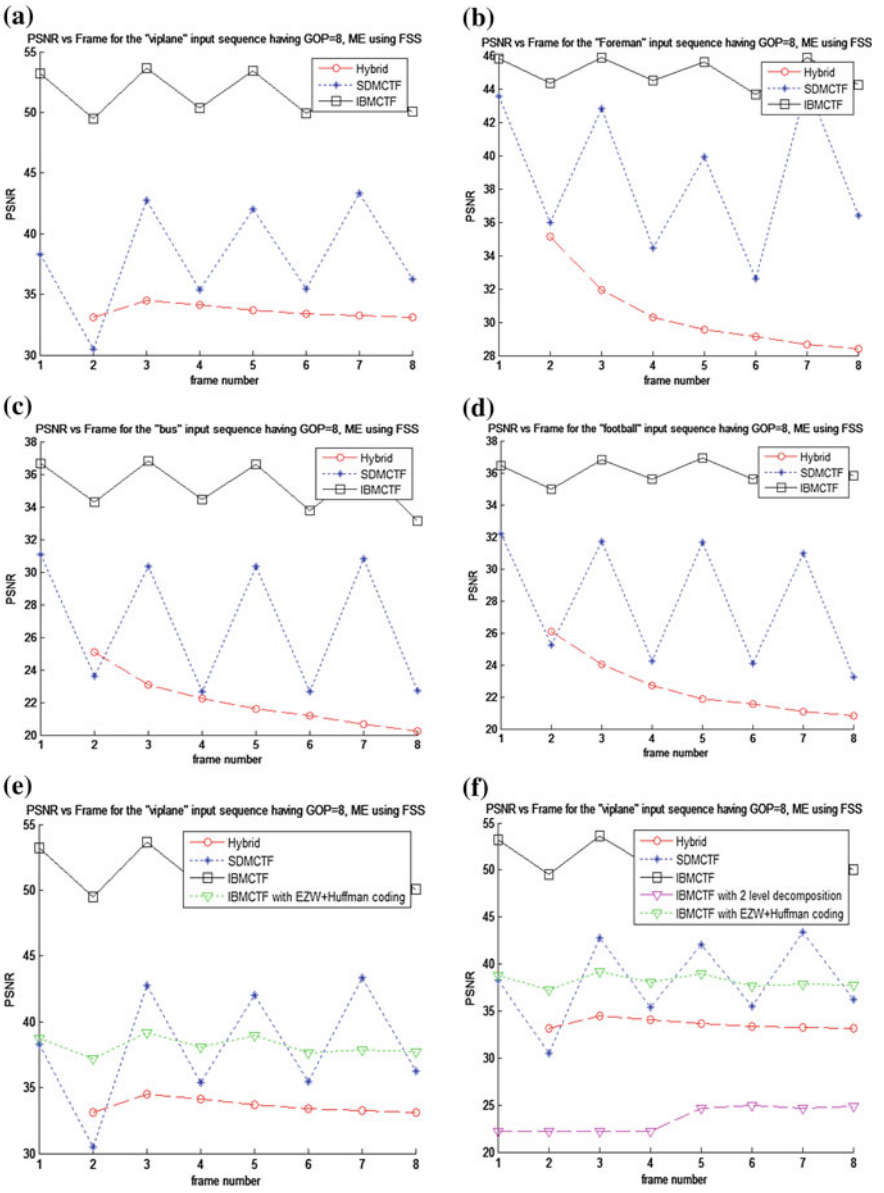


Fig. 7.22 Performance (PSNR in dB) comparison between IB-MCTF, SD-MCTF and Hybrid model for the different input sequences: **a** 'viplane', **b** 'foreman', **c** 'Bus', **d** 'football', **e** IB-MCTF with entropy coding for 'viplane' sequence, **f** IB-MCTF with 2-level temporal decomposition for 'viplane' sequence

Table 7.1 Performance (PSNR in dB) comparison between IB-MCTF and SD-MCTF

Frame no.	IB-MCTF				SD-MCTF			
	Viplane	Foreman	Bus	Football	Viplane	Foreman	Bus	Football
1	53.20	45.80	36.64	36.42	38.31	43.53	31.05	32.17
2	49.51	44.34	34.30	34.96	30.47	35.95	23.60	25.22
3	53.62	45.88	36.79	36.83	42.77	42.83	30.35	31.73
4	50.38	44.50	34.46	35.59	35.36	34.44	22.64	24.24
5	53.43	45.58	36.58	36.92	42.01	39.92	30.36	31.64
6	49.92	43.67	33.76	35.60	35.46	32.61	22.66	24.08
7	52.18	45.87	36.36	37.26	43.35	44.24	30.81	30.95
8	50.04	44.25	33.15	35.79	36.24	36.39	22.70	23.22

4 HF frames. For the second level, 4 LF frames are the inputs and then we get 2 LF and 2 HF frames, at the third level, we get 1 LF and 1 HF frame. Maximum numbers of levels in temporal direction are three, because the GOP size is eight. After the 3rd level decomposition in temporal direction we have 1 LF and 7 HF frames. Entropy coding applies to those frames. We tested the simulated architectures with "viplane", "foreman", "bus", and "football" video sequences (Figs. 7.21 and 7.22). Results shows that SVC based on IB-MCTF provides the better PSNR and bit rate. Performance evaluation is given in Table 7.1.

7.5 Conclusions

The present framework is concerned with implementation of IB-MCTF. It has been demonstrated that IB-MCTF performs better than SD-MCTF as there exists greater freedom in choosing various ME schemes for different sub-bands. But the memory requirement and the computational complexity increase as we go down higher levels of spatial and temporal scalability. For a two layer (base layer + one enhancement layer) encoding, IB-MCTF requires more computation than SD-MCTF. As the level of decomposition increases the computational requirements increases by the order of 2^{2n} where n is the level of decomposition. Hence we can conclude that the IB-MCTF is preferred in research oriented fields where the quality of the received video data is more significant. In domains of medical imaging and distant medical applications where the quality of the video is significantly more important than the end-to-end delay, IB-MCTF is better than SD-MCTF.

References

1. Ohm, J.R.: Advances in scalable video coding. Proc. IEEE **93**, 42–56 (2005)
2. Advanced Video Coding for Generic Audiovisual Services.: ITU-T Rec. H.264 and ISO/IEC 14496–10 (MPEG-4 AVC), ITU-T and ISO/IEC JTC 1, Version 1: May 2003, Version 2: May 2004, Version 3: March 2005, Version 4: September 2005, Version 5 and Version 6: June 2006, Version 7: April 2007, Version 8 (including SVC extension): Consented in July 2007
3. ITU-T Rec. & ISO/IEC 14496–10 AVC.: Advanced video coding for generic audiovisual services, version 3 (2005)
4. Schwarz, H., et al.: Technical Description of the HHI proposal for SVC CE1, ISO/IEC JTC1/WG11, Doc. m11244, Palma de Mallorca, Spain. October 2004
5. Reichel, J., Schwarz, H., Wien, M.: Scalable video coding joint draft 6. Joint Video Team, Doc. JVT-S201, Geneva, Switzerland, April 2006
6. Coding of audiovisual objectsPart 10.: Advanced video coding. International Standards Organisation/International Electrotechnical Commission (ISO/IEC), ISO/IEC14 496-10 (identical to ITU-T Recommendation H.264)
7. Richardson, I.E.G.: H.264 and MPEG-4 Video Compression Video Coding for Next generation Multimedia. Wiley, West Sussex (2003)
8. Andreopoulos, Y., Van Der Schaar, M., Munteanu, A., Barbarien, J., Schelkens, P., Cornelis, J.: Complete-to-overcomplete discrete wavelet transforms for scalable video coding with MCTF. In: Proceedings of the SPIE/IEEE Visual Communications and Image Processing, pp. 719–731 (2003)
9. Andreopoulos, Y., Munteanu, A., Barbarien, J., Van der Schaar, M., Cornelis, J., Schelkens, P.: In-band motion compensated temporal filtering. Signal Process.: Image Commun. **19**, 653–673 (2004)
10. Vanhoof, B., Peón, M., Lafruit, G., Bormans, J., Nachtergaele, L., Bolsens, I.: A scalable architecture for MPEG-4 wavelet quantization. J. VLSI Signal Process.-Syst. Signal, Image Video Technol. **23**(1) (1999) (Special Issue on Implementation of MPEG-4 Multimedia Codecs)
11. Wang, B., Loo, K.K., Yip, P.Y., Siyau M.F.: A simplified scalable wavelet video codec with MCTF structure. In: International Conference on Digital Telecommunications (ICDT'06) 29–31 August 2006. doi:10.1109/ICDT.2006.11
12. Andreopoulos, Y., van der Schaar, M., Munteanu, A., Barbarien, J., Schelkens, P., Cornelis, J.: Fully-scalable wavelet video coding using in-band motion-compensated temporal filtering. In: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, pp. III-417-III-420 (2003)
13. Park, H.W., Kim, H.-S.: Motion estimation using low-band-shift method for wavelet-based moving-picture coding. IEEE Trans. Image Process. **9**(4), 577–587 (2000)
14. Daubechies, I., Sweldens, W.: Factoring wavelet transforms into lifting steps. J. Fourier Anal. Apple **4**(3), 247–269 (1998)
15. Vaidyanathan, P.P.: Multirate Systems and Filter Banks, 1st edn. Prentice Hall Inc., Englewood Cliffs (1992)
16. Strang, G., Nguyen, T.: Wavelets and Filter Banks. Wellesley-Cambridge Press (1996), ISBN: 0-9614088-7-1
17. Barjatya, A.: Block matching algorithms for motion estimation. In: Technical Report, Utah State University (2004)
18. Ohm, J.R.: Three-dimensional sub-band coding with motion compensation. IEEE Trans. Image Process. **3**, 559–571 (1994)
19. Ohm, J.R.: Motion-compensated 3D sub-band coding with multiresoultion representation of motion parameters. Proc. ICIP 1994 **3**, 250–254 (1994)
20. Choi, S., Woods, J.: Motion compensated 3D sub-band coding of video. IEEE Trans. Image Process. **8**, 155–167 (1999)
21. Schelkens, P., Andreopoulos, Y., Barbarien, J., Clerckx, T., Verdicchio, F., Munteanu, A., van der Schaar M.: A comparative study of scalable video coding schemes utilizing wavelet technology. In: Proceedings of SPIE Photonics East, Wavelet Applications in Industrial Processing, vol. 5266, pp. 147–156 Providence (2004)

22. Barbarien, J., et al.: Scalable motion vector coding. In: Proceedings of the International Conference on Image Processing 2004 (ICIP'04), vol. 2, 24–27 (October 2004)
23. Andra, K., Chakrabarti, C., Acharya, T.: A VLSI architecture for lifting-based forward and inverse wavelet transform. In: IEEE Transactions on Signal Processing, **50**(4), April 2002
24. Weeks, M., Bayoumi, M.A.: Three-dimensional discrete wavelet transform architectures. IEEE Trans. Signal Process., **50**(8), 2050–2063 (2002). doi:10.1109/TSP.2002.800402
25. Zhang, C., Wang, C., Omair Ahmad, M.: A pipeline VLSI architecture for high-speed computation of the 1-D discrete wavelet transform. IEEE Trans. Circuits Syst., **57**(10), 2729–2740 (2010). doi:10.1109/TCSI.2010.2046974
26. Das, A., Hazra, A., Banerjee, S.: An efficient architecture for 3-D discrete wavelet transform. IEEE Trans. Circuits Syst. Video Technol., **20**, 286–296 (2010). doi:10.1109/TCSVT.2009.2031551
27. Mohanty B.K., Meher, P.K.: Parallel and pipeline architectures for high-throughput computation of multilevel 3-D DWT. IEEE Trans. Circuits Syst. Video Technol., **20**(9), 1200–1209 (2010). doi:10.1109/TCSVT.2010.2056950
28. Mohanty, B.K., Mahajan, A., Meher, P.K.: Area- and power-efficient architecture for high-throughput implementation of lifting 2-D DWT. IEEE Trans. Circuits Syst. II: Express Br., **59**(7), 434–438 (2012). doi:10.1109/TCSII.2012.2200169
29. Zhang, W., Jiang, Z., Gao, Z., Liu, Y.: An efficient VLSI architecture for lifting-based discrete wavelet transform. IEEE Trans. Circuits Syst. II: Express Br. **59**(3), 158–162 (2012)

Chapter 8
Forward Plans

The previous five chapters have recorded major contributions of the present research, which may be enumerated as follows:

1. Design and implementation of efficient VLSI architecture for Fast Three Step search motion estimation algorithm.
2. Development of a parallel VLSI architecture for successive elimination algorithm.
3. Design and implementation of a high performance motion estimation architecture based on diamond search algorithm and one-bit transformation.
4. Development of a new algorithm for motion estimation based on pixel truncation. Also designed an efficient architecture for the same.
5. Implementation of scalable video coding based on in-band motion compensated temporal filtering (IBMCTF) through lifting based DWT.

Possible extensions of the research undertaken so far can be contemplated as follows.

8.1 SoC Based Design for SVC

Sooner or later, wired as well as wireless communication will be characterized by transmission of video frames over variable bandwidth channels. The need of the video signals to be displayed on the entire gamut of devices ranging from mobile cellphones to ultra-high definition television systems calls for extensive adoption of the principle of scalable video coding (SVC) in video transmission [1]. Scalable video coding is deemed applicable to even the latest video coding standard, viz. High Efficiency Video Coding (HEVC) or H.265 [2–4]. To manage the complexity of H.265, FPGA devices can be used as co-processors or accelerators to achieve a real-time encoder/decoder system. Major FPGA vendors such as Altera and Xilinx currently offer powerful System-on-Chip (SoC) devices (Altera's Arria V and Cylone V series, and Xilinx's Zynq-7000). These SoC platforms appear to be feasible approaches for development, prototyping, and production. For, they provide both flexibility and performance. Based on simple calculation, the 4 K encoding process can be split into

I. Chakrabarti et al., *Motion Estimation for Video Coding*,
Studies in Computational Intelligence 590, DOI 10.1007/978-3-319-14376-7_8

four parallel processing pipelines, with two SoC chips performing the processing in parallel. The newly introduced SoC devices have dual ARM processors and FPGA fabric, so that motion estimation, motion compensation, and inter prediction blocks can be implemented in the FPGA fabric. Moreover, DWT can be implemented in the FPGA DSP area, and Syntax assembly and entropy coding can be handled by the ARM core. As the complexity of video codec algorithm increases, such SoC FPGAs are likely to be effective means of implementation.

8.2 Scalable Extension of HEVC

Figure 8.1 shows the structure of the encoder for the proposed scalable video codec pertaining to the case of input HDTV sequence, and examples of bit-stream composition for different spatial and temporal resolution. The decoder can automatically

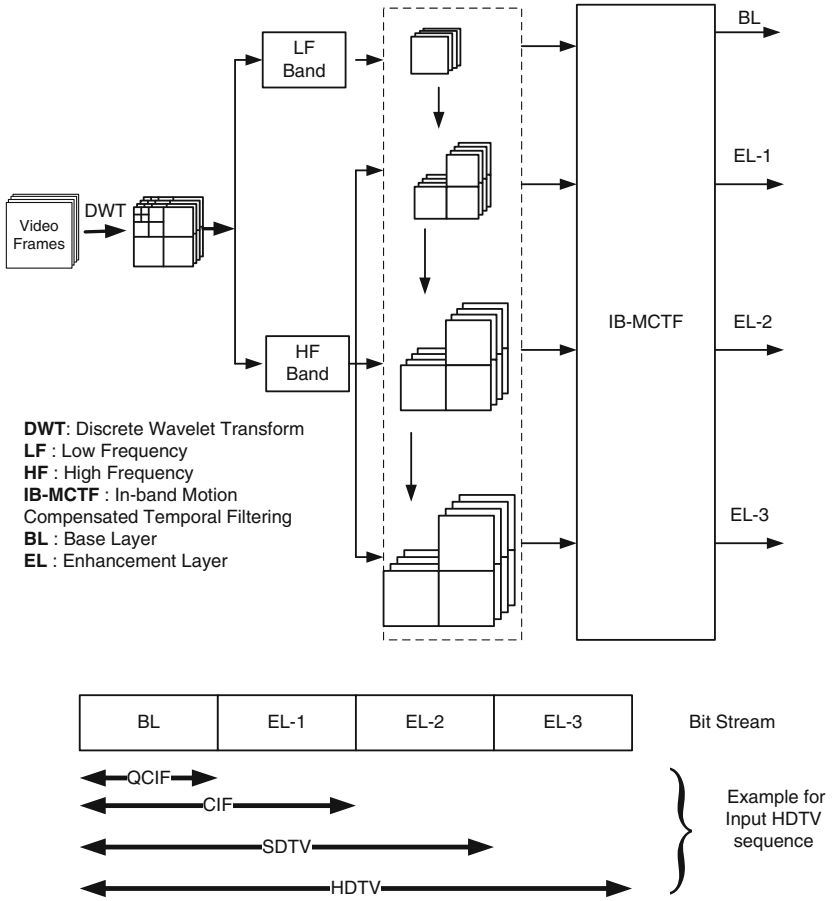


Fig. 8.1 The encoding structure that performs open-loop framework for SVC using IBMCTF

acquire necessary bit-streams right from the lowest level QCIF to the highest level HDTV depending on the network environment and the spatial-temporal display requirement of the user terminal. For spatial changes, if the input video frame is of an SD resolution (704 × 576) and the client wants to display the video in CIF (352 × 288) at its terminal, the combination bit-stream between QCIF and CIF will be necessary. It is known that better video quality can be obtained from increasingly higher resolution. Hence, there is a need to use more video information from upper resolution levels to reconstruct the picture [1, 5–7]. For temporal variation, one may have to drop some frames in order to suit the network transmission condition. In this scenario, certain bit-streams will be selected from the different resolution to compose the output.

References

1. Ohm, J.R.: Advances in scalable video coding. Proc. of the IEEE **93**, 42–56 (2005)
2. Sullivan, G.J., Ohm, J.R., Jin Han, W., Wiegand, T.: Overview of the high efficiency video coding (HEVC) standard. IEEE Trans. Circuits Syst. Video Technol. **22**(12), 1649–1668 (2012)
3. Ohm, J.R., Sullivan, G.J., Schwarz, H., Tan, T.K., Wiegand, T.: Comparison of the coding efficiency of video coding standardsincluding high efficiency video coding (HEVC). IEEE Trans. Circuits Syst. Video Technol. **22**(12), 1669–1684 (2012)
4. Bossen, F., Bross, B., Sühring, K., Flynn, D.: HEVC complexity and implementation analysis. IEEE Trans. Circuits Syst. Video Technol. **22**(12), 1685–1696 (2012)
5. Andreopoulos, Y., van der Schaar, M., Munteanu, A., Barbarien, J., Schelkens, P., Cornelis, J.: Fully-scalable wavelet video coding using in-band motion compensated temporal filtering. In: Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'03), pp. III-417–III-20, **3**, 6–10 April 2003. doi:10.1109/ICASSP.2003.1199500
6. Andreopoulos, Y., Munteanu, A., Barbarien, J., van der Schaar, M., Cornelis, J., Schelkens, P.: In-band motion compensated temporal filtering. Signal Process. Image Commun. **19**, 653–673 (2004)
7. Wang, B., Loo, K.K., Yip, P.Y., Siyau, M.F.: A simplified scalable wavelet video codec with MCTF structure. In: Proceedings of International Conference on Digital Telecommunications (ICDT'06), 29–31 August 2006. doi:10.1109/ICDT.2006.11

Appendix A
Matlab Programs

A.1 Program for Fast Three Step Search Algorithm

```
xx=imread('foreman13.jpg');
yy=imread('foreman14.jpg');
[H W]=size(xx);
mri1= zeros(H+16,W+16);
mri2= zeros(H,W);
searchpoint=zeros(9,2);
blok=zeros(64,2);
madvalue=zeros(9,1);
motionvector=zeros((H/16)*(W/16),2);
ptemp=0;
a3=zeros(H,W);
a2=uint8(zeros(H,W));

    for sr=1:H
      for sc=1:W
    mri1(sr+7,sc+7)=xx(sr,sc,1);           %previous frame
    mri2(sr,sc)=yy(sr,sc,1);                 %current frame
      end
    end
%###start dividing current frame in to 16X16 blocks
Blocks=zeros(16,16,(H/16)*(W/16));
fr=1;
for row1=0:16:H-16
   for col1=0:16:W-16
        for row=1:16
            for col=1:16
               Blocks(row,col,fr)=mri2((row+row1),(col+col1));
            end
```


I. Chakrabarti et al., *Motion Estimation for Video Coding*,
Studies in Computational Intelligence 590, DOI 10.1007/978-3-319-14376-7

```
        end
        fr = fr+1;
    end
end
%#### Finish dividing current frame in to 16X16 blocks #
%###  start loading search window  ###
buff=zeros(30,32,(H/16)*(W/16));
fr=1;
for row1=0:16:H-16
   for col1=0:16:W-16
        for row=1:30
            for col=1:32
              buff(row,col,fr)=mri1((row+row1),(col+col1));
            end
        end
        fr = fr+1;
    end
end
for frameno=1:(H/16)*(W/16)
 % for loop on "frameno" started from here
   bi=0;
   % frameno=30;

    %1st step  address starts  from here

    [srow,scol]=searchaddr(frameno);
    [crow,ccol]=currentaddr(frameno);
      spx=3;
      for sp1=1:3:7
          spy=3;
        for sp2=1:3
          searchpoint(sp1+sp2-1,1)=srow+spx;
          searchpoint(sp1+sp2-1,2)=scol+spy;
          spy=spy+4;
        end
            spx=spx+4;
      end

%first step starts from here
j=3;
for i=1:3

      for mad1st=5:6
          searx=searchpoint(mad1st,1);
          seary=searchpoint(mad1st,2);
```

```
          mad1=0;
         for a=0:15
             for b=0:15
 t1=buff(a+searx-(floor((frameno-1)/8)*16),
 b+seary-(rem((frameno-1),8)*16),frameno);
 t2=Blocks(a+crow-(floor((frameno-1)/8)*16),
 b+ccol-(rem((frameno-1),8)*16),frameno);
                 ptemp=procelem(t1,t2);
                 mad1=mad1+ptemp;
             end
         end
         madvalue(mad1st,1)=mad1;
         bi=bi+1;
   end
         mad1st=8;
         searx=searchpoint(mad1st,1);
         seary=searchpoint(mad1st,2);
          mad1=0;
    for a=0:15
         for b=0:15
  t1=buff(a+searx-(floor((frameno-1)/8)*16),
  b+seary-(rem((frameno-1),8)*16),frameno);
  t2=Blocks(a+crow-(floor((frameno-1)/8)*16),
  b+ccol-(rem((frameno-1),8)*16),frameno);
             ptemp=procelem(t1,t2);
             mad1=mad1+ptemp;
         end
    end
    madvalue(mad1st,1)=mad1;
     bi=bi+1;

if((madvalue(6,1)< madvalue(5,1))&&
(madvalue(8,1)< madvalue(5,1)))
% main if condition starts from here
            mad1st=9;
            searx=searchpoint(mad1st,1);
            seary=searchpoint(mad1st,2);
           mad1=0;
         for a=0:15
             for b=0:15
   t1=buff(a+searx-(floor((frameno-1)/8)*16),
   b+seary-(rem((frameno-1),8)*16),frameno);
   t2=Blocks(a+crow-(floor((frameno-1)/8)*16),
   b+ccol-(rem((frameno-1),8)*16),frameno);
                 ptemp=procelem(t1,t2);
```

```
                    mad1=mad1+ptemp;
                end
            end
                      madvalue(mad1st,1)=mad1;
                       bi=bi+1;

                if((madvalue(6,1)<madvalue(9,1))&&
                  (madvalue(6,1)<madvalue(8,1)))
                    s2row=searchpoint(6,1);
                    s2col=searchpoint(6,2);
                elseif((madvalue(8,1)<madvalue(9,1))&&
                (madvalue(8,1)<madvalue(9,1)))
                    s2row=searchpoint(8,1);
                    s2col=searchpoint(8,2);
                else
                     s2row=searchpoint(9,1);
                    s2col=searchpoint(9,2);
                end
    elseif((madvalue(6,1)<madvalue(5,1))&&
    (madvalue(5,1)<madvalue(8,1)))
                    mad1st=3;
                    searx=searchpoint(mad1st,1);
                    seary=searchpoint(mad1st,2);
                      mad1=0;
                for a=0:15
                    for b=0:15
        t1=buff(a+searx-(floor((frameno-1)/8)*16),
        b+seary-(rem((frameno-1),8)*16),frameno);
        t2=Blocks(a+crow-(floor((frameno-1)/8)*16),
        b+ccol-(rem((frameno-1),8)*16),frameno);
                       ptemp=procelem(t1,t2);
                      mad1=mad1+ptemp;
                    end
                end
                       madvalue(mad1st,1)=mad1;
                        bi=bi+1;

                    if(madvalue(6,1)<madvalue(3,1))
                        s2row=searchpoint(6,1);
                        s2col=searchpoint(6,2);
                    else
                        s2row=searchpoint(3,1);
                        s2col=searchpoint(3,2);
                    end
    elseif((madvalue(8,1)<madvalue(5,1))&&
```

```
        (madvalue(5,1)<madvalue(6,1)))
                        mad1st=7;
                        searx=searchpoint(mad1st,1);
                        seary=searchpoint(mad1st,2);
                         mad1=0;
                for a=0:15
                    for b=0:15
       t1=buff(a+searx-(floor((frameno-1)/8)*16),
       b+seary-(rem((frameno-1),8)*16),frameno);
      t2=Blocks(a+crow-(floor((frameno-1)/8)*16),
      b+ccol-(rem((frameno-1),8)*16),frameno);
                         ptemp=procelem(t1,t2);
                         mad1=mad1+ptemp;
                    end
                end
                    madvalue(mad1st,1)=mad1;
                     bi=bi+1;

                if(madvalue(8,1)<madvalue(7,1))
                    s2row=searchpoint(8,1);
                    s2col=searchpoint(8,2);
                else
                    s2row=searchpoint(7,1);
                    s2col=searchpoint(7,2);
                end
elseif((madvalue(5,1)<madvalue(6,1))&&
  (madvalue(5,1)<madvalue(8,1)))
    if((madvalue(5,1)<madvalue(6,1))&&
      (madvalue(6,1)<madvalue(8,1)))
              mad1st=2;
              searx=searchpoint(mad1st,1);
              seary=searchpoint(mad1st,2);
                mad1=0;
           for a=0:15
               for b=0:15
        t1=buff(a+searx-(floor((frameno-1)/8)*16),
        b+seary-(rem((frameno-1),8)*16),frameno);
       t2=Blocks(a+crow-(floor((frameno-1)/8)*16),
       b+ccol-(rem((frameno-1),8)*16),frameno);
                    ptemp=procelem(t1,t2);
                    mad1=mad1+ptemp;
                end
           end
              madvalue(mad1st,1)=mad1;
               bi=bi+1;
```

```
    if(madvalue(2,1)<madvalue(5,1))
            mad1st=1;
            searx=searchpoint(mad1st,1);
            seary=searchpoint(mad1st,2);
              mad1=0;
        for a=0:15
            for b=0:15
  t1=buff(a+searx-(floor((frameno-1)/8)*16),
  b+seary-(rem((frameno-1),8)*16),frameno);
  t2=Blocks(a+crow-(floor((frameno-1)/8)*16),
  b+ccol-(rem((frameno-1),8)*16),frameno);
                    ptemp=procelem(t1,t2);
                    mad1=mad1+ptemp;
              end
        end
               madvalue(mad1st,1)=mad1;
                bi=bi+1;
            if(madvalue(2,1)<madvalue(1,1))
                s2row=searchpoint(2,1);
                s2col=searchpoint(2,2);
            else
                s2row=searchpoint(1,1);
                s2col=searchpoint(1,2);
            end
       else
            s2row=searchpoint(5,1);
            s2col=searchpoint(5,2);
    end
else
           mad1st=4;
           searx=searchpoint(mad1st,1);
           seary=searchpoint(mad1st,2);
           mad1=0;
        for a=0:15
            for b=0:15
   t1=buff(a+searx-(floor((frameno-1)/8)*16),
     b+seary-(rem((frameno-1),8)*16),frameno);
     t2=Blocks(a+crow-(floor((frameno-1)/8)*16),
      b+ccol-(rem((frameno-1),8)*16),frameno);
               ptemp=procelem(t1,t2);
               mad1=mad1+ptemp;
            end
        end
            madvalue(mad1st,1)=mad1;
             bi=bi+1;
```

```
            if(madvalue(4,1)<madvalue(5,1))
                mad1st=1;
                searx=searchpoint(mad1st,1);
                seary=searchpoint(mad1st,2);
                mad1=0;
                    for a=0:15
                        for b=0:15
           t1=buff(a+searx-(floor((frameno-1)/8)*16),
           b+seary-(rem((frameno-1),8)*16),frameno);
     t2=Blocks(a+crow-(floor((frameno-1)/8)*16),
     b+ccol-(rem((frameno-1),8)*16),frameno);
                           ptemp=procelem(t1,t2);
                           mad1=mad1+ptemp;
                        end
                    end
                      madvalue(mad1st,1)=mad1;
                       bi=bi+1;
                    if(madvalue(4,1)<madvalue(1,1))
                        s2row=searchpoint(4,1);
                        s2col=searchpoint(4,2);
                    else
                        s2row=searchpoint(1,1);
                        s2col=searchpoint(1,2);
                    end
                else
                            s2row=searchpoint(5,1);
                            s2col=searchpoint(5,2);
            end
    end
else  %if((madvalue(5,1)< madvalue(8,1))&&
(madvalue(8,1)<madvalue(6,1)))
       mad1st=4;
       searx=searchpoint(mad1st,1);
       seary=searchpoint(mad1st,2);
       mad1=0;
    for a=0:15
        for b=0:15
       t1=buff(a+searx-(floor((frameno-1)/8)*16),
         b+seary-(rem((frameno-1),8)*16),frameno);
      t2=Blocks(a+crow-(floor((frameno-1)/8)*16),
       b+ccol-(rem((frameno-1),8)*16),frameno);
             ptemp=procelem(t1,t2);
             mad1=mad1+ptemp;
        end
```

```
        end
            madvalue(mad1st,1)=mad1;
             bi=bi+1;
         if(madvalue(4,1)<madvalue(5,1))
            mad1st=1;
            searx=searchpoint(mad1st,1);
            seary=searchpoint(mad1st,2);
            mad1=0;
                for a=0:15
                    for b=0:15
            t1=buff(a+searx-(floor((frameno-1)/8)*16),
            b+seary-(rem((frameno-1),8)*16),frameno);
            t2=Blocks(a+crow-(floor((frameno-1)/8)*16),
            b+ccol-(rem((frameno-1),8)*16),frameno);
                       ptemp=procelem(t1,t2);
                       mad1=mad1+ptemp;
                    end
                end
                  madvalue(mad1st,1)=mad1;
                  bi=bi+1;
                if(madvalue(4,1)<madvalue(1,1))
                    s2row=searchpoint(4,1);
                    s2col=searchpoint(4,2);
                else
                    s2row=searchpoint(1,1);
                    s2col=searchpoint(1,2);
                end
            else
                        s2row=searchpoint(5,1);
                        s2col=searchpoint(5,2);
        end
    end
           j=j-1;
          spx=0;
          for sp1=1:3:7
              spy=0;
              for sp2=1:3
                  searchpoint(sp1+sp2-1,1)=s2row+spx-j;
                  searchpoint(sp1+sp2-1,2)=s2col+spy-j;
              spy=spy+j;
              end
             spx=spx+j;
          end
end
      motionvector(frameno,1,frame)=s2row-(srow+7);
```

```
      motionvector(frameno,2,frame)=s2col-(scol+7);
      blok(frameno,1)=frameno;
       blok(frameno,2)=bi;
 end % for loop on "frameno" end here
```

A.2 Program for Successive Elimination Algorithm

```
xx=imread('b11.bmp');
yy=imread('b12.bmp');
[H W]=size(xx);
mri1= zeros(H+32,W+32);
mri2= zeros(H, W);
motionvector=zeros((H/16)*(W/16),2);
colarray=zeros(48,1);
ptemp=0;
count1=0;
a3=zeros(H, W);
a2=uint8(zeros(H, W));
    for sr=1:H
        for sc=1:W
            a3(sr,sc)=128;
        end
    end
    for sr=1:128
      for sc=1:128
            mri1(sr+16,sc+16)=xx(sr,sc,1); %previous frame
            mri2(sr,sc)=yy(sr,sc,1);        %current frame
      end
    end

%# start dividing current frame in to 16X16 blocks #
Blocks=zeros(16,16,(H/16)*(W/16));
fr=1;
for row1=0:16:H-16
   for col1=0:16:W-16
        for row=1:16
            for col=1:16
      Blocks(row,col,fr)=mri2((row+row1),(col+col1));
            end
        end
        fr = fr+1;
```

```
    end
end
%# Finish dividing of current frame in to 16X16 blocks #
%# start search window loading #
buff=zeros(48,48,(H/16)*(W/16));
fr=1;
for row1=0:16:H-16
   for col1=0:16:W-16
        for row=1:48
            for col=1:48
              buff(row,col,fr)=mri1((row+row1),(col+col1));
            end
        end
        fr = fr+1;
    end
end
for frameno=1:(H/16)*(W/16)
% frameno for loop started from here
       mad1=0;
       count=0;
     refsum=0;
           for a=1:16
              for b=1:16
                  t1=buff(a,b,frameno);
                  t2=Blocks(a,b,frameno);
                  refsum=refsum+t2;
                  ptemp=procelem(t1,t2);
                  mad1=mad1+ptemp;
              end
            end
            count=count+1;
            srow=0;
            scol=0;
           cur_min_sad=mad1;
           for k=1:48
               colarray(k,1)=0;
               for l=1:16
         colarray(k,1)=colarray(k,1)+ buff(l,k,frameno);
               end
           end
           temp1=0;
           for m=1:16
           temp1=temp1+colarray(m,1);
           end
          sr=1;
```

```
        for  sc=2:31
             temp12=temp1-colarray(sc-1,1)+colarray(sc+15,1);
             temp1=temp12;
             sad_sn=abs(refsum-temp12);
          if (cur_min_sad>sad_sn)
              mad1=0;
               for a=1:16
                  for b=1:16
                      t1=buff(a,b+sc-1,frameno);
                      t2=Blocks(a,b,frameno);
                      ptemp=procelem(t1,t2);
                      mad1=mad1+ptemp;
                  end
               end
                count=count+1;
              if (cur_min_sad>=mad1)
                  cur_min_sad=mad1;
                  srow=sr;
                  scol=sc;
              else
              end
          else
          end
          mad1=0;
        end
    for sr=2:30
        for k=1:46
              colarray(k,1)=0;
              for l=sr:sr+16
         colarray(k,1)=colarray(k,1)+ buff(l,k,frameno);
              end
        end

        for  sc=1:32
            temp1=0;
          for m=sc:sc+16
          temp1=temp1+colarray(m,1);
          end
             sad_sn=abs(refsum-temp1);
          if (cur_min_sad>=sad_sn)
              mad1=0;
               for a=1:16
                  for b=1:16
                      t1=buff(a+sr-1,b+sc-1,frameno);
                      t2=Blocks(a,b,frameno);
```

```
                    ptemp=procelem(t1,t2);
                    mad1=mad1+ptemp;
                end
             end
              count=count+1;
            if (cur_min_sad>=mad1)
                cur_min_sad=mad1;
                srow=sr;
                scol=sc;
            else
            end
        else
        end
        mad1=0;
      end
  end
  count1=count1+count;
  motionvector(frameno,1)=srow-16;
  motionvector(frameno,2)=scol-16;
end %for loop on " frameno" end here
```

Appendix B
Verilog Modules

B.1 Simulation Program for Fast Three Step Search Algorithm

```
‘timescale 1ns / 1ps
module ftss(clk,start, mad12,mad22,mad32,memno1,
memadr1,memno2,memadr2,memno3,memadr3,pe_en);
    input clk;
//  input restart;
 input start;
 output [15:0] mad12;
 output [15:0] mad22;
 output [15:0] mad32;
 output [7:0]memno1; output [7:0]memno2; output [7:0]memno3;
 output [7:0]memadr1; output[7:0]memadr2;output[7:0]memadr3;
 output pe_en;
     reg [7:0] memmod1[0:8][0:63];
 reg [7:0] memmod2 [0:8][0:63];
 reg step11; reg step12; reg step21; reg step22;
 reg step31; reg step32;   reg storen;
 reg pe_en; reg pe1en; reg pe2en;  reg restart;
 reg [7:0]ram [0:1][0:8];
 reg signed [7:0]motionvect [0:1][0:15];
 reg [15:0]madval [0:8];
 reg [7:0]memr[0:63][0:63];
 reg [7:0]searchmem[0:79][0:79];
 reg [7:0]curmem[0:15][0:15];
 reg [7:0]memry[0:4095];
 reg [7:0]memry1[0:4095];
 reg [15:0] mad12;  reg [15:0] mad22;
 reg [15:0] mad32;   reg [15:0] mad11;
```


I. Chakrabarti et al., *Motion Estimation for Video Coding*,
Studies in Computational Intelligence 590, DOI 10.1007/978-3-319-14376-7

```
reg [15:0] mad21;  reg [15:0] mad31;
reg [15:0] temp1;  reg [15:0] temp2;
reg [15:0] temp3;reg [15:0] madtemp;
reg [7:0]memno1;  reg [7:0]memno2;
reg [7:0]memno3;   reg [7:0]memadr1;
reg [7:0]memadr2;  reg [7:0]memadr3;
reg [7:0]roi;  reg [7:0]roi2;
reg [7:0]coj;reg [7:0]roi3; reg [7:0]coj3;
reg [7:0]coj2; reg [7:0]dist;
reg [7:0]rrt; reg [7:0]rct;reg [7:0]p;
integer i,j,l,l1,ir,i1,i2,si,sj,ssi,ssj,
m,n,i3,j3,k,a,b,c,d,mv1,mv2,mv3;
integer gli,j1,ar,ac,ar1,ar2,ar3,ac1,ac2,
ac3,s2temp,s2temp2,blockno,madloc,s3temp;

always@(posedge start)
begin
restart=1'b0;
l=0;l1=0;gli=0;
blockno=0;
$readmemh("127.txt",memry,0,4095);
for(i=0;i<=63;i=i+1)
for(j=0;j<=63;j=j+1)
begin
memr[i][j]=memry[l];
l=l+1;
end
for(si=0;si<=79;si=si+1)
for(sj=0;sj<=79;sj=sj+1)
searchmem[si][sj]=128;
$readmemh("128.txt",memry1,0,4095);
for(ssi=0;ssi<=63;ssi=ssi+1)
for(ssj=0;ssj<=63;ssj=ssj+1)
begin
searchmem[ssi+7][ssj+7]=memry1[l1];
l1=l1+1;
end
for(i=0;i<=8;i=i+1)
madval[i]=16'd65535;
end

always@(posedge restart)
begin
step11=1'b0; step12=1'b0;   step21=1'b0;
step22=1'b0; step31=1'b0;   step32=1'b0;
```

```
mad12[15:0]=16'd0000;
mad22[15:0]=16'd0000;
mad32[15:0]=16'd0000;
l=0;l1=0;m=0;n=0;
ir=0;c=0;d=0;
ar=(blockno/4)*16;
ac=(blockno%4)*16;
ar1=0;ar2=0;ar3=0;
ac1=0;ac2=0;ac3=0;
for(i1=3;i1<=11;i1=i1+4)
for(i2=3;i2<=11;i2=i2+4)
begin
ram[0][ir]=i1;
ram[1][ir]=i2;
ir=ir+1;
end
for(i=0;i<=15;i=i+1)
for(j=0;j<=15;j=j+1)
curmem[i][j]=memr[i+ar][j+ac];
for(a=ar;a<=ar+29;a=a+3)
for(b=ac;b<=ac+30;b=b+3)
begin
if(b<(ac+15))
begin
memmod1[0][m]=searchmem[a][b];
memmod1[1][m]=searchmem[a][b+1];
memmod1[2][m]=searchmem[a][b+2];
memmod1[3][m]=searchmem[a+1][b];
memmod1[4][m]=searchmem[a+1][b+1];
memmod1[5][m]=searchmem[a+1][b+2];
   memmod1[6][m]=searchmem[a+2][b];
   memmod1[7][m]=searchmem[a+2][b+1];
memmod1[8][m]=searchmem[a+2][b+2];
m=m+1;
end
else if(b==(ac+15))
begin
memmod1[0][m]=searchmem[a][b];
memmod1[3][m]=searchmem[a+1][b];
memmod1[6][m]=searchmem[a+1][b];
m=m+1;
memmod2[0][n]=searchmem[a][b];
memmod2[1][n]=searchmem[a][b+1];
memmod2[2][n]=searchmem[a][b+2];
memmod2[3][n]=searchmem[a+1][b];
```

```
memmod2[4][n]=searchmem[a+1][b+1];
memmod2[5][n]=searchmem[a+1][b+2];
   memmod2[6][n]=searchmem[a+2][b];
   memmod2[7][n]=searchmem[a+2][b+1];
memmod2[8][n]=searchmem[a+2][b+2];
n=n+1;
end
else if((b>(ac+15))&&(b<(ac+30)))
begin
memmod2[0][n]=searchmem[a][b];
memmod2[1][n]=searchmem[a][b+1];
memmod2[2][n]=searchmem[a][b+2];
memmod2[3][n]=searchmem[a+1][b];
memmod2[4][n]=searchmem[a+1][b+1];
memmod2[5][n]=searchmem[a+1][b+2];
   memmod2[6][n]=searchmem[a+2][b];
   memmod2[7][n]=searchmem[a+2][b+1];
memmod2[8][n]=searchmem[a+2][b+2];
n=n+1;
end
else if(b==(ac+30))
begin
memmod2[0][n]=searchmem[a][b];
memmod2[3][n]=searchmem[a+1][b];
memmod2[6][n]=searchmem[a+2][b];
n=n+1;
end
end
for(si=0;si<=8;si=si+1)
for(sj=0;sj<=63;sj=sj+1)
begin
$display("memory value memmod 1 [%d][%d]=%h",
si,sj,memmod1[si][sj]);
$display("memory value memmod 2 [%d][%d]=%h",
si,sj,memmod2[si][sj]);
end
if((blockno>>2)==0)
begin
mv1=4;
mv2=5;
mv3=7;
end
else
begin
if((motionvect[0][blockno-1]<0)&&
```

```
(motionvect[1][blockno-1]<0))
begin
mv1=1;
mv2=3;
mv3=4;
end
else if((motionvect[0][blockno-1]<0)&&
(motionvect[1][blockno-1]>0))
begin
mv1=1;
mv2=4;
mv3=5;
end
else if((motionvect[0][blockno-1]>0)&&
(motionvect[1][blockno-1]<0))
begin
mv1=3;
mv2=4;
mv3=7;
end
else
begin
mv1=4;
mv2=5;
mv3=7;
end
end
end

always@clk
begin
gli=gli+1;
if((gli/2)==0)
restart=1'b1;
else if (((gli/2)>0)&&((gli/2)<=256))
//for step 1,first phase
begin
restart=1'b0;
step11=clk;
end
else if((gli/2)==257)
begin
pe_en=1'b1;
step11=1'b0;
 //compare the 3 search locations and find the next
```

```
//one or two search locations
 end
    else if(((gli/2)>257)&&((gli/2)<=514))
     //for step 1, second phase
 begin
 pe_en=1'b0;
 step12=clk;
 end
 else if((gli/2)==515)
 begin
 storen=1'b1;        // starts 2 nd step
 dist[7:0]=8'b00000010;
 end
 else if(((gli/2)>515)&&((gli/2)<=771))
 //for step 2,first phase
 begin
 storen=1'b0;
 step12=1'b0;
 step21=clk;
 end
 else if((gli/2)==772)
 pe_en=1'b1;
 else if(((gli/2)>772)&&((gli/2)<=1029))
 //for step 2,second phase
 begin
 pe_en=1'b0;
 step21=1'b0;
 step22=clk;
 end
 else if((gli/2)==1030)
 begin
 storen=1'b1;
 // starts 3rd  step
 dist[7:0]=8'b00000001;
 end
 else if(((gli/2)>1030)&&((gli/2)<=1286))
 //for step 3,first phase
 begin
 storen=1'b0;
 step22=1'b0;
 step31=clk;
 end
 else if((gli/2)==1287)
 pe_en=1'b1;
 else if(((gli/2)>1287)&&((gli/2)<=1543))
```

```
 //for step 3,second phase
 begin
 pe_en=1'b0;
 step31=1'b0;
 step32=clk;
 end
 else
 begin
 step32=1'b0;
 if(blockno<16)
 //find the motion vector.
 begin
 gli=0;
 motionvect[0][blockno]=rrt-7;
 motionvect[1][blockno]=rct-7;
 blockno=blockno+1;
 for(j=0;j<=16;j=j+1)
 for(i=0;i<=1;i=i+1)
 $display("motion vector values :
 motion vectors[%d][%d]=%d",i,j,motionvect[i][j]);
 end
 end
 end

 always@(posedge step11 or posedge step21 or posedge step31)
 begin
adgen1(ar1+ram[0][mv1],ac1+ram[1][mv1],memno1,memadr1);
if(ac1+ram[1][mv1]<=15)
procelem1(curmem[ar1][ac1],memmod1[memno1][memadr1],mad11);
 else
procelem1(curmem[ar1][ac1],memmod2[memno1][memadr1],mad11);
 temp1=mad11+mad12;
 mad12=temp1;
 if((ac1<15)&&(ar1<=15))
 ac1=ac1+1;
 else if((ac1==15)&&(ar1<15))
 begin
 ac1=0;
 ar1=ar1+1;
 end
 else
 begin
 ac1=0;
 ar1=0;
 madval[4]=mad12;
```

```
 end
 end

always@(posedge step11 or posedge step21 or posedge step31)
begin
adgen2(ar2+ram[0][mv2],ac2+ram[1][mv2],memno2,memadr2);
if(ac2+ram[1][mv2]<=15)
procelem2(curmem[ar2][ac2],memmod1[memno2][memadr2],mad21);
else
procelem2(curmem[ar2][ac2],memmod2[memno2][memadr2],mad21);
 temp2=mad21+mad22;
 mad22=temp2;
 if((ac2<15)&&(ar2<=15))
 ac2=ac2+1;
 else if((ac2==15)&&(ar2<15))
 begin
 ac2=0;
 ar2=ar2+1;
 end
 else
 begin
 ac2=0;
 ar2=0;
 madval[5]=mad22;
 end
 end

always@(posedge step11 or posedge step21 or posedge step31)
begin
adgen3(ar3+ram[0][mv3],ac3+ram[1][mv3],memno3,memadr3);
if(ac3+ram[1][mv3]<=15)
procelem3(curmem[ar3][ac3],memmod1[memno3][memadr3],mad31);
else
procelem3(curmem[ar3][ac3],memmod2[memno3][memadr3],mad31);
 temp3=mad31+mad32;
 mad32=temp3;
 if((ac3<15)&&(ar3<=15))
 ac3=ac3+1;
 else if((ac3==15)&&(ar3<15))
 begin
 ac3=0;
 ar3=ar3+1;
 end
 else
 begin
```

```
 ac3=0;
 ar3=0;
 madval[7]=mad32;
 end
 end
always@(posedge pe_en)
begin
mad12[15:0]=16'd0000;
mad22[15:0]=16'd0000;
mad32[15:0]=16'd0000;
ar1=0; ac1=0;ar3=0;
ar2=0; ac2=0;ac3=0;
if ((mv1==4)&&(mv2==5))
begin
if((madval[5]<madval[4])&&(madval[7]<madval[4]))
begin
roi=ram[0][8];
coj=ram[1][8];
s2temp=8;
pe1en=1'b1;
pe2en=1'b0;
end
else if((madval[5]<madval[4])&&(madval[4]<madval[7]))
begin
roi=ram[0][2];
coj=ram[1][2];
s2temp=2;
pe1en=1'b1;
pe2en=1'b0;
end
else if((madval[7]<madval[4])&&(madval[4]<madval[5]))
begin
roi=ram[0][6];
coj=ram[1][6];
s2temp=6;
pe1en=1'b1;
pe2en=1'b0;
end
else
begin
roi=ram[0][0];
coj=ram[1][0];
s2temp=0;
roi2=ram[0][1];
coj2=ram[1][1];
```

```
s2temp2=1;
pe1en=1'b1;
pe2en=1'b1;
roi3=ram[0][3];
coj3=ram[1][3];
s3temp=3;
end
end
else if ((mv1==1)&&(mv2==3))
begin
if((madval[1]<madval[4])&&(madval[3]<madval[4]))
begin
roi=ram[0][0];
coj=ram[1][0];
s2temp=0;
pe1en=1'b1;
pe2en=1'b0;
end
else if((madval[1]<madval[4])&&(madval[4]<madval[3]))
begin
roi=ram[0][2];
coj=ram[1][2];
s2temp=2;
pe1en=1'b1;
pe2en=1'b0;
end
else if((madval[3]<madval[4])&&(madval[4]<madval[3]))
begin
roi=ram[0][6];
coj=ram[1][6];
s2temp=6;
pe1en=1'b1;
pe2en=1'b0;
end
else
begin
roi=ram[0][5];
coj=ram[1][5];
s2temp=5;
roi2=ram[0][7];
coj2=ram[1][7];
s2temp2=7;
pe1en=1'b1;
pe2en=1'b1;
roi3=ram[0][8];
```

```
coj3=ram[1][8];
s3temp=8;
end
end
else if ((mv1==1)&&(mv2==4))
begin
if((madval[1]<madval[4])&&(madval[5]<madval[4]))
begin
roi=ram[0][2];
coj=ram[1][2];
s2temp=2;
pe1en=1'b1;
pe2en=1'b0;
end
else if((madval[1]<madval[4])&&(madval[4]<madval[5]))
begin
roi=ram[0][0];
coj=ram[1][0];
s2temp=0;
pe1en=1'b1;
pe2en=1'b0;
end
else if((madval[5]<madval[4])&&(madval[4]<madval[1]))
begin
roi=ram[0][8];
coj=ram[1][8];
s2temp=8;
pe1en=1'b1;
pe2en=1'b0;
end
else
begin
roi=ram[0][3];
coj=ram[1][3];
s2temp=3;
roi2=ram[0][6];
coj2=ram[1][6];
s2temp2=6;
pe1en=1'b1;
pe2en=1'b1;
roi3=ram[0][7];
coj3=ram[1][7];
s3temp=7;
end
end
```

```
else if ((mv1==3)&&(mv2==4))
begin
if((madval[3]<madval[4])&&(madval[7]<madval[4]))
begin
roi=ram[0][6];
coj=ram[1][6];
s2temp=6;
pe1en=1'b1;
pe2en=1'b0;
end
else if((madval[3]<madval[4])&&(madval[4]<madval[7]))
begin
roi=ram[0][60];
coj=ram[1][0];
s2temp=0;
pe1en=1'b1;
pe2en=1'b0;
end
else if((madval[7]<madval[4])&&(madval[4]<madval[3]))
begin
roi=ram[0][8];
coj=ram[1][8];
s2temp=8;
pe1en=1'b1;
pe2en=1'b0;
end
else
begin
roi=ram[0][1];
coj=ram[1][1];
s2temp=1;
roi2=ram[0][2];
coj2=ram[1][2];
s2temp2=2;
pe1en=1'b1;
pe2en=1'b1;
roi3=ram[0][5];
coj3=ram[1][5];
s3temp=5;
end
end
end
always@(posedge step12 or posedge step22 or posedge step32)
begin
adgen1(ar1+roi,ac1+coj,memno1,memadr1);
```

```
if(ac1+coj<=15)
procelem1(curmem[ar1][ac1],memmod1[memno1][memadr1],mad11);
else
procelem1(curmem[ar1][ac1],memmod2[memno1][memadr1],mad11);
temp1=mad11+mad12;
 mad12=temp1;
 if((ac1<15)&&(ar1<=15))
 ac1=ac1+1;
 else if((ac1==15)&&(ar1<15))
 begin
 ac1=0;
 ar1=ar1+1;
 end
 else
 begin
 ac1=0;
 ar1=0;
 madval[s2temp]=mad12;
 end
 end
always@(posedge step12 or posedge step22 or posedge step32)
begin
adgen2(ar2+roi2,ac2+coj2,memno2,memadr2);
if(ac2+coj2<=15)
procelem2(curmem[ar2][ac2],memmod1[memno2][memadr2],mad21);
else
procelem2(curmem[ar2][ac2],memmod2[memno2][memadr2],mad21);
temp2=mad21+mad22;
mad22=temp2;
 if((ac2<15)&&(ar2<=15))
 ac2=ac2+1;
 else if((ac2==15)&&(ar2<15))
 begin
 ac2=0;
 ar2=ar2+1;
 end
 else
 begin
 ac2=0;
 ar2=0;
 madval[s2temp2]=mad22;
 end
 end
always@(posedge step12 or posedge step22 or posedge step32)
begin
```

```
adgen3(ar3+roi3,ac3+coj3,memno3,memadr3);
if(ac3+coj3<=15)
procelem3(curmem[ar3][ac3],memmod1[memno3][memadr3],mad31);
else
procelem3(curmem[ar3][ac3],memmod2[memno3][memadr3],mad31);
 temp3=mad31+mad32;
 mad32=temp3;
 if((ac3<15)&&(ar3<=15))
 ac3=ac3+1;
 else if((ac3==15)&&(ar3<15))
 begin
 ac3=0;
 ar3=ar3+1;
 end
 else
 begin
 ac3=0;
 ar3=0;
 madval[s3temp]=mad32;
 end
 end

 always@(posedge storen)
 begin
 madtemp=16'd65535;
 for(p=0;p<=8;p=p+1)
 begin
 if(madval[p]<madtemp)
 begin
 madtemp=madval[p];
 madloc=p;
 end
 end
 rrt=ram[0][madloc];
 rct=ram[1][madloc];
 ram[0][0]=rrt-dist;
 ram[1][0]=rct-dist;
 ram[0][1]=rrt-dist;
 ram[1][1]=rct;
 ram[0][2]=rrt-dist;
 ram[1][2]=rct+dist;
 ram[0][3]=rrt;
 ram[1][3]=rct-dist;
 ram[0][4]=rrt;
 ram[1][4]=rct;
```

```
 ram[0][5]=rrt;
 ram[1][5]=rct+dist;
 ram[0][6]=rrt+dist;
 ram[1][6]=rct-dist;
 ram[0][7]=rrt+dist;
 ram[1][7]=rct;
 ram[0][8]=rrt+dist;
 ram[1][8]=rct+dist;
 mad12[15:0]=16'd0000;
 mad22[15:0]=16'd0000;
 mad32[15:0]=16'd0000;
 ar1=0; ac1=0;
 ar2=0; ac2=0;
 ar3=0; ac3=0;
 end
task adgen1(input [7:0] row1, col1,
output [7:0] modno1, modadr1);
   begin
if(col1>15)
col1=col1-15;
modno1=((row1%3)*3+(col1%3));
   modadr1=((row1/3)*6+(col1/3));
   end
endtask
task adgen2(input [7:0] row2, col2,
output [7:0] modno2, modadr2);
   begin
if(col2>15)
col2=col2-15;
modno2=((row2%3)*3+(col2%3));
   modadr2=((row2/3)*6+(col2/3));
   end
endtask
task adgen3(input [7:0] row3, col3,
output [7:0] modno3, modadr3);
   begin
if(col3>15)
col3=col3-15;
modno3=((row3%3)*3+(col3%3));
   modadr3=((row3/3)*6+(col3/3));
   end
endtask
task procelem1;
    input [7:0] curpix1;
    input [7:0] searchpix1;
```

```
 output [15:0] mad1;
   begin
if (curpix1 < searchpix1)
begin
mad1[7:0]=searchpix1-curpix1;
mad1[15:8]=8'b00000000;
end
else
begin
mad1[7:0]=curpix1-searchpix1;
mad1[15:8]=8'b00000000;
end
end
endtask
task procelem2;
   input [7:0] curpix2;
   input [7:0] searchpix2;
output [15:0] mad2;
   begin
if (curpix2 < searchpix2)
begin
mad2[7:0]=searchpix2-curpix2;
mad2[15:8]=8'b00000000;
end
else
begin
mad2[7:0]=curpix2-searchpix2;
mad2[15:8]=8'b00000000;
end
end
endtask
task procelem3;
   input [7:0] curpix3;
   input [7:0] searchpix3;
output [15:0] mad3;
   begin
if (curpix3 < searchpix3)
begin
mad3[7:0]=searchpix3-curpix3;
mad3[15:8]=8'b00000000;
end
else
begin
mad3[7:0]=curpix3-searchpix3;
mad3[15:8]=8'b00000000;
```

```
end
end
endtask
endmodule
```

B.2 Simulation Program for Successive Elimination Algorithm

```
‘timescale 1ns / 1ps
module successive1(clk,start,cur_min_sad,
tesum1,madsum,ar1,contcol,controw,spix1,
spix2,spix3,spix4,spix5,spix6,spix7);
    input clk;
 input start;
 output [15:0] cur_min_sad;
 output [15:0]tesum1,madsum;
 output[7:0] ar1,contcol,controw,spix1,spix2,
 spix3,spix4,spix5,spix6,spix7;
 reg [7:0] buffer1 [0:15][0:47];
 reg [7:0] buffer2 [0:15][0:47];
 reg [7:0] buffer3 [0:15][0:47];
 reg restart; reg cont;reg cont1;
 reg signed [7:0]motionvect [0:1][0:15];
 reg [7:0]memr[0:63][0:63];
 reg [7:0]searchmem[0:95][0:95];
 reg [7:0]curblock[0:15][0:15];
 reg [7:0]memry[0:4095];
 reg [7:0]memry1[0:4095];
 reg [15:0] temad,itemad1,itemad2,
 itemad3,itemad4,itesum1,itesum2,itesum3,itesum4;
 reg [15:0]temad0,temad1,temad2,temad3,temad4,
 temad5,temad6,temad15;
 reg [15:0] temad7,temad8,temad9,temad10,temad11,
 temad12,temad13,temad14;
 reg [15:0] tesum1,refsum,cur_min_sad,madsum;
 reg [7:0]contar1,spix0,spix1,spix2,spix3,spix4,
 spix5,spix6,spix7;
 reg [7:0]spix8,spix9,spix10,spix11,spix12,spix13,
 spix14,spix15;
 reg [7:0]roi,roi2,coj,roi3,coj3,controw,contcol;
 reg [7:0]coj2;reg [7:0]dist;   reg [7:0]rrt;
```

```
reg [7:0]rct;reg [7:0]p,ar1;
integer i,j,l,l1,ir,i1,i2,si,sj,ssi;
integer ssj,m,n,i3,j3,k,a,b,c,d;
integer gli,j1,ar,ac,blockno;
integer lp1,lp2,lp3,pq,ai;

always@(posedge start)
//INITIALIZING ALL MEMORYS WITH PIXELS
begin
restart=1'b0;
l=0;l1=0;
cont1=1'b1;
blockno=0;
$readmemh("127.txt",memry,0,4095);
for(i=0;i<=63;i=i+1)
for(j=0;j<=63;j=j+1)
begin
memr[i][j]=memry[l];
l=l+1;
end
for(si=0;si<=95;si=si+1)
for(sj=0;sj<=95;sj=sj+1)
searchmem[si][sj]=8'b10000000;
$readmemh("128.txt",memry1,0,4095);
for(ssi=0;ssi<=63;ssi=ssi+1)
for(ssj=0;ssj<=63;ssj=ssj+1)
begin
searchmem[ssi+16][ssj+16]=memry1[l1];
l1=l1+1;
end
end

always@(posedge restart)
begin
refsum[15:0]=16'h0000;
temad[15:0]=16'h0000;
tesum1[15:0]=16'h0000;
cur_min_sad[15:0]=16'hffff;
controw=0;
contcol=0;
contar1=0;
cont=1'b0;
l=0;l1=0;m=0;n=0;
ir=0; pq=0;ai=0;
ar=(blockno/4)*16;
```

```
ac=(blockno%4)*16;
ar1=0;lp1=0;lp2=0;lp3=0;
for(i=0;i<=15;i=i+1)
for(j=0;j<=15;j=j+1)
begin
curblock[j][i]=memr[i+ar][j+ac];
refsum=refsum+memr[i+ar][j+ac];
end
for(a=ar;a<=ar+47;a=a+1)
begin
for(b=ac;b<=ac+47;b=b+1)
begin
if(b<(ac+16))
begin
buffer1[lp1][m]=searchmem[a][b];
lp1=lp1+1;
end
else if(b>=(ac+16)&& b<(ac+32))
begin
buffer2[lp2][n]=searchmem[a][b];
lp2=lp2+1;
end
else if((b>=(ac+32))&&(b<(ac+48)))
begin
buffer3[lp3][pq]=searchmem[a][b];
lp3=lp3+1;
end
end
n=n+1;
m=m+1;
pq=pq+1;
lp1=0;lp2=0;lp3=0;
end
for(si=0;si<=15;si=si+1)
for(sj=0;sj<=47;sj=sj+1)
begin
$display("memory value buffer 1 [%d][%d]=%h",
si,sj,buffer1[si][sj]);
$display("memory value buffer 2 [%d][%d]=%h",
si,sj,buffer2[si][sj]);
$display("memory value buffer 3 [%d][%d]=%h",
si,sj,buffer3[si][sj]);
end
end
```

```
 // BEGINING OF CONTROL BLOCK

 always@(posedge clk)
 begin
 if(cont1==1'b1)
 begin
 restart=1'b1;
 cont1=1'b0;
 end
 else
 begin
 restart=1'b0;
 if (cont==1'b0)
begin
 if((controw<=31)&&(contcol<=31))
begin
searchpixgen(ar1+controw,contcol,spix0,spix1,
spix2,spix3,spix4,spix5, spix6,spix7,spix8,spix9,spix10,
spix11,spix12,spix13,spix14,spix15);
itesum1=spix0+spix1+spix2+spix3;
itesum2=spix4+spix5+spix6+spix7;
itesum3=spix8+spix9+spix10+spix11;
itesum4=spix12+spix13+spix14+spix15;
tesum1=tesum1+itesum1+itesum2+itesum3+itesum4;
if(ar1<15)
ar1=ar1+1;
  else
begin
 if (tesum1>refsum)
madsum=tesum1-refsum;
 else
 madsum=refsum-tesum1;

if(cur_min_sad[15:0] > madsum[15:0])
 cont=1'b1;
else
begin
ar1=0;
contcol=contcol+1;
tesum1[15:0]=16'h0000;
end
end
end
else if((contcol>31)&&(controw<32))
 begin
```

```
 contcol=0;
 controw=controw+1;
 end
else if(controw>31)
 begin
 contcol=0;
 controw=0;
 cont1=1'b1;
 motionvect[0][blockno]=rrt-16;
 motionvect[1][blockno]=rct-16;
 blockno=blockno+1;
 for(j=0;j<=blockno;j=j+1)
 for(i=0;i<=1;i=i+1)
$display("motion vector values :motion vectors
[%d][%d]=%d",i,j,motionvect[i][j]);
 end
 end
 else
begin
searchpixgen(contar1+controw,contcol,spix0,
spix1,spix2, spix3,spix4,spix5, spix6,spix7,
spix8,spix9,spix10,spix11,spix12,spix13,
 spix14,spix15);

procelem0(curblock[0][contar1],spix0,temad0);
procelem1(curblock[1][contar1],spix1,temad1);
procelem2(curblock[2][contar1],spix2,temad2);
procelem3(curblock[3][contar1],spix3,temad3);
procelem4(curblock[4][contar1],spix4,temad4);
procelem5(curblock[5][contar1],spix5,temad5);
procelem6(curblock[6][contar1],spix6,temad6);
procelem7(curblock[7][contar1],spix7,temad7);
procelem8(curblock[8][contar1],spix8,temad8);
procelem9(curblock[9][contar1],spix9,temad9);
procelem10(curblock[10][contar1],spix10,temad10);
procelem11(curblock[11][contar1],spix11,temad11);
procelem12(curblock[12][contar1],spix12,temad12);
procelem13(curblock[13][contar1],spix13,temad13);
procelem14(curblock[14][contar1],spix14,temad14);
procelem15(curblock[15][contar1],spix15,temad15);
itemad1=temad0+temad1+temad2+temad3;
itemad2=temad4+temad5+temad6+temad7;
itemad3=temad8+temad9+temad10+temad11;
itemad4=temad12+temad13+temad14+temad15;
 temad=temad+itemad1+itemad2+itemad3+itemad4;
```

```
if(contar1<15)
contar1=contar1+1;
 else
 begin
 if (cur_min_sad > temad)
 begin
 cur_min_sad=temad;
 rrt=controw;
 rct=contcol;
 end
contar1=0;
ar1=0;
cont=1'b0;
contcol=contcol+1;
temad[15:0]=16'h0000;
 end
 end

end
end


task searchpixgen( input [7:0] row1, col1,
output [7:0]searchpix0, searchpix1,searchpix2,
searchpix3,searchpix4,  searchpix5,
searchpix6,searchpix7,searchpix8, searchpix9,
 searchpix10,searchpix11,searchpix12,searchpix13,
  searchpix14,searchpix15);
   begin
if(col1<16)
searchpix0=buffer1[col1][row1];
else if (col1>=16 && col1<32)
searchpix0=buffer2[col1-16][row1];
else
searchpix0=buffer3[col1-32][row1];
   if(col1+1<=15)
searchpix1=buffer1[col1+1][row1];
else if (col1+1>=16 && col1+1<32)
searchpix1=buffer2[(col1+1)-16][row1];
else
searchpix1=buffer3[(col1+1)-32][row1];
   if(col1+2<=15)
searchpix2=buffer1[col1+2][row1];
else if (col1+2>=16 && col1+2<32)
searchpix2=buffer2[(col1+2)-16][row1];
```

```
else
searchpix2=buffer3[(col1+2)-32][row1];
   if(col1+3<=15)
searchpix3=buffer1[col1+3][row1];
else if (col1+3>=16 && col1+3<32)
searchpix3=buffer2[(col1+3)-16][row1];
else
searchpix3=buffer3[(col1+3)-32][row1];
   if(col1+4<=15)
searchpix4=buffer1[col1+4][row1];
else if (col1+4>=16 && col1+4<32)
searchpix4=buffer2[(col1+4)-16][row1];
else
searchpix4=buffer3[(col1+4)-32][row1];
   if(col1+5<=15)
searchpix5=buffer1[col1+5][row1];
else if (col1+5>=16 && col1+5<32)
searchpix5=buffer2[(col1+5)-16][row1];
else
searchpix5=buffer3[(col1+5)-32][row1];
   if(col1+6<=15)
searchpix6=buffer1[col1+6][row1];
else if (col1+6>=16 && col1+6<32)
searchpix6=buffer2[(col1+6)-16][row1];
else
searchpix6=buffer3[(col1+6)-32][row1];
   if(col1+7<=15)
searchpix7=buffer1[col1+7][row1];
else if (col1+7>=16 && col1+7<32)
searchpix7=buffer2[(col1+7)-16][row1];
else
searchpix7=buffer3[(col1+7)-32][row1];
   if(col1+8<=15)
searchpix8=buffer1[col1+8][row1];
else if (col1+8>=16 && col1+8<32)
searchpix8=buffer2[(col1+8)-16][row1];
else
searchpix8=buffer3[(col1+8)-32][row1];
   if(col1+9<=15)
searchpix9=buffer1[col1+9][row1];
else if (col1+9>=16 && col1+9<32)
searchpix9=buffer2[(col1+9)-16][row1];
else
searchpix9=buffer3[(col1+9)-32][row1];
   if(col1+10<=15)
```

```
searchpix10=buffer1[col1+10][row1];
else if (col1+10>=16 && col1+10<32)
searchpix10=buffer2[(col1+10)-16][row1];
else
searchpix10=buffer3[(col1+10)-32][row1];
   if(col1+11<=15)
searchpix11=buffer1[col1+11][row1];
else if (col1+11>=16 && col1+11<32)
searchpix11=buffer2[(col1+11)-16][row1];
else
searchpix11=buffer3[(col1+11)-32][row1];
   if(col1+12<=15)
searchpix12=buffer1[col1+12][row1];
else if (col1+12>=16 && col1+12<32)
searchpix12=buffer2[(col1+12)-16][row1];
else
searchpix12=buffer3[(col1+12)-32][row1];
   if(col1+13<=15)
searchpix13=buffer1[col1+13][row1];
else if (col1+13>=16 && col1+13<32)
searchpix13=buffer2[(col1+13)-16][row1];
else
searchpix13=buffer3[(col1+13)-32][row1];
   if(col1+14<=15)
searchpix14=buffer1[col1+14][row1];
else if (col1+14>=16 && col1+14<32)
searchpix14=buffer2[(col1+14)-16][row1];
else
searchpix14=buffer3[(col1+14)-32][row1];
   if(col1+15<=15)
searchpix15=buffer1[col1+15][row1];
else if (col1+15>=16 && col1+15<32)
searchpix15=buffer2[(col1+15)-16][row1];
else
searchpix15=buffer3[(col1+15)-32][row1];
   end
endtask

task procelem0(input [7:0] cpix0, spix0, output [15:0] temad0);
begin
if (cpix0 < spix0)
begin
temad0[7:0]=spix0-cpix0;
temad0[15:8]=8'b00000000;
end
```

```
else
begin
temad0[7:0]=cpix0-spix0;
temad0[15:8]=8'b00000000;
end
end
endtask
task procelem1(input [7:0] cpix1,spix1,
output [15:0] temad1);
begin
if (cpix1 < spix1)
begin
temad1[7:0]=spix1-cpix1;
temad1[15:8]=8'b00000000;
end
else
begin
temad1[7:0]=cpix1-spix1;
temad1[15:8]=8'b00000000;
end
end
endtask
task procelem2(input [7:0] cpix2,spix2,
 output [15:0] temad2);
begin
if (cpix2 < spix2)
begin
temad2[7:0]=spix2-cpix2;
temad2[15:8]=8'b00000000;
end
else
begin
temad2[7:0]=cpix2-spix2;
temad2[15:8]=8'b00000000;
end
end
endtask
task procelem3(input [7:0] cpix3,spix3,
output [15:0] temad3);
begin
if (cpix3 < spix3)
begin
temad3[7:0]=spix3-cpix3;
temad3[15:8]=8'b00000000;
end
```

```
else
begin
temad3[7:0]=cpix3-spix3;
temad3[15:8]=8'b00000000;
end
end
endtask
task procelem4(input [7:0] cpix4,spix4,
output [15:0] temad4);
begin
if (cpix4 < spix4)
begin
temad4[7:0]=spix4-cpix4;
temad4[15:8]=8'b00000000;
end
else
begin
temad4[7:0]=cpix4-spix4;
temad4[15:8]=8'b00000000;
end
end
endtask
task procelem5(input [7:0] cpix5, spix5,
output [15:0] temad5);
begin
if (cpix5 < spix5)
begin
temad5[7:0]=spix5-cpix5;
temad5[15:8]=8'b00000000;
end
else
begin
temad5[7:0]=cpix5-spix5;
temad5[15:8]=8'b00000000;
end
end
endtask
task procelem6(input [7:0] cpix6,spix6,
 output [15:0] temad6);
begin
if (cpix6 < spix6)
begin
temad6[7:0]=spix6-cpix6;
temad6[15:8]=8'b00000000;
end
```

```
else
begin
temad6[7:0]=cpix6-spix6;
temad6[15:8]=8'b00000000;
end
end
endtask
task procelem7(input [7:0] cpix7,spix7,
 output [15:0] temad7);
begin
if (cpix7 < spix7)
begin
temad7[7:0]=spix7-cpix7;
temad7[15:8]=8'b00000000;
end
else
begin
temad7[7:0]=cpix7-spix7;
temad7[15:8]=8'b00000000;
end
end
endtask
task procelem8(input [7:0] cpix8,spix8,
output [15:0] temad8);
begin
if (cpix8 < spix8)
begin
temad8[7:0]=spix8-cpix8;
temad8[15:8]=8'b00000000;
end
else
begin
temad8[7:0]=cpix8-spix8;
temad8[15:8]=8'b00000000;
end
end
endtask
task procelem9(input [7:0] cpix9,spix9,
output [15:0] temad9);
begin
if (cpix9 < spix9)
begin
temad9[7:0]=spix9-cpix9;
temad9[15:8]=8'b00000000;
end
```

```
else
begin
temad9[7:0]=cpix9-spix9;
temad9[15:8]=8'b00000000;
end
end
endtask
task procelem10(input [7:0] cpix10,spix10,
output [15:0] temad10);
begin
if (cpix10 < spix10)
begin
temad10[7:0]=spix10-cpix10;
temad10[15:8]=8'b00000000;
end
else
begin
temad10[7:0]=cpix10-spix10;
temad10[15:8]=8'b00000000;
end
end
endtask
task procelem11(input [7:0] cpix11,spix11,
output [15:0] temad11);
begin
if (cpix11 < spix11)
begin
temad11[7:0]=spix11-cpix11;
temad11[15:8]=8'b00000000;
end
else
begin
temad11[7:0]=cpix11-spix11;
temad11[15:8]=8'b00000000;
end
end
endtask
task procelem12(input [7:0] cpix12,spix12,
output [15:0] temad12);
begin
if (cpix12 < spix12)
begin
temad12[7:0]=spix12-cpix12;
temad12[15:8]=8'b00000000;
end
```

```
else
begin
temad12[7:0]=cpix12-spix12;
temad12[15:8]=8'b00000000;
end
end
endtask
task procelem13(input [7:0] cpix13,spix13,
output [15:0] temad13);
begin
if (cpix13 < spix13)
begin
temad13[7:0]=spix13-cpix13;
temad13[15:8]=8'b00000000;
end
else
begin
temad13[7:0]=cpix13-spix13;
temad13[15:8]=8'b00000000;
end
end
endtask
task procelem14(input [7:0] cpix14,spix14,
output [15:0] temad14);
begin
if (cpix14 < spix14)
begin
temad14[7:0]=spix14-cpix14;
temad14[15:8]=8'b00000000;
end
else
begin
temad14[7:0]=cpix14-spix14;
temad14[15:8]=8'b00000000;
end
end
endtask
task procelem15(input [7:0] cpix15,spix15,
output [15:0] temad15);
begin
if (cpix15 < spix15)
begin
temad15[7:0]=spix15-cpix15;
temad15[15:8]=8'b00000000;
end
```

```
else
begin
temad15[7:0]=cpix15-spix15;
temad15[15:8]=8'b00000000;
end
end
endtask
endmodule
```

Index

I. Chakrabarti et al., *Motion Estimation for Video Coding*,
Studies in Computational Intelligence 590, DOI 10.1007/978-3-319-14376-7